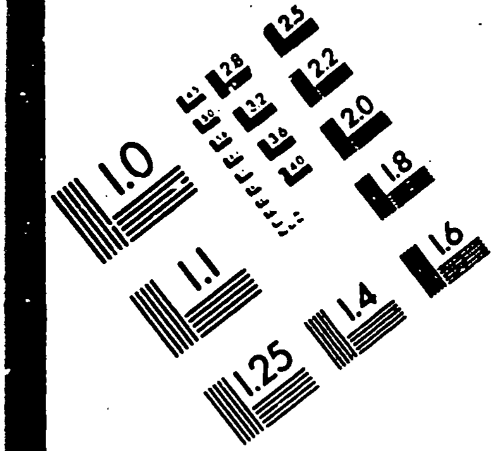
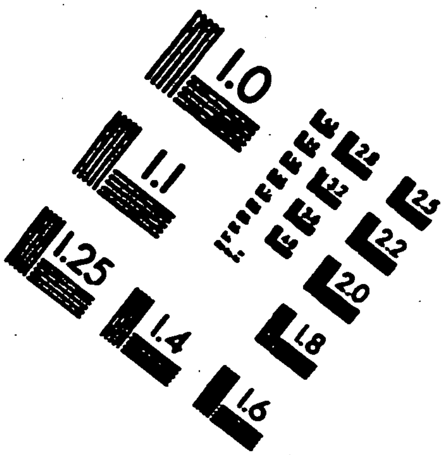
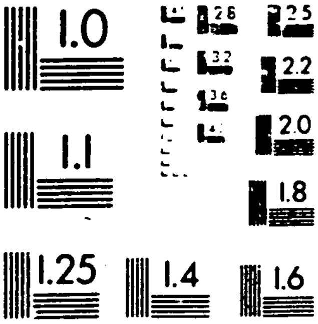
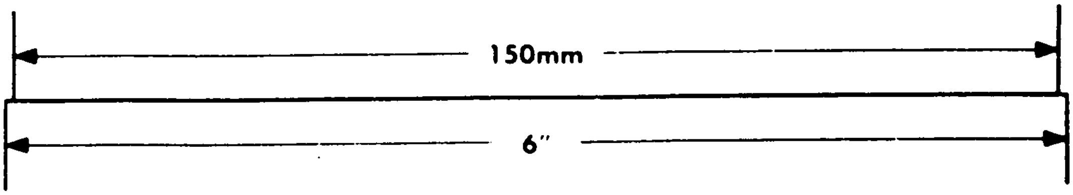
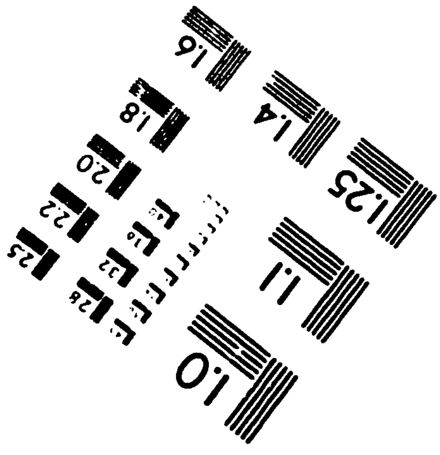


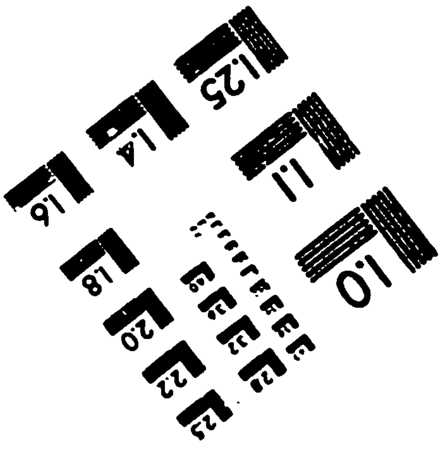
IMAGE EVALUATION TEST TARGET (MT-3)











PHOTOGRAPHIC SCIENCES CORPORATION
770 BASKET ROAD
P.O. BOX 338
WEBSTER, NEW YORK 14580
(716) 265-1600

microunity

Zeus System Architecture

MicroUnity

Craig Hansen
Chief Architect

MicroUnity Systems Engineering, Inc.
475 Potrero Avenue
Sunnyvale, CA 94086-4118
Phone: 408.734.8100
Fax: 408.734.8136
email: craig@microunity.com
http://www.microunity.com

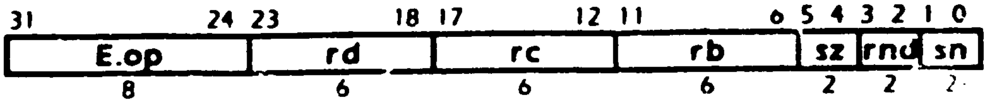
E.MULX.I.32.N	Ensemble multiply extract immediate signed quadlets nearest
E.MULX.I.32.Z	Ensemble multiply extract immediate signed quadlets zero
E.MULX.I.64.C	Ensemble multiply extract immediate signed octlets ceiling
E.MULX.I.64.F	Ensemble multiply extract immediate signed octlets floor
E.MULX.I.64.N	Ensemble multiply extract immediate signed octlets nearest
E.MULX.I.64.Z	Ensemble multiply extract immediate signed octlets zero
E.MULX.I.C.8.C	Ensemble multiply extract immediate complex bytes ceiling
E.MULX.I.C.8.F	Ensemble multiply extract immediate complex bytes floor
E.MULX.I.C.8.N	Ensemble multiply extract immediate complex bytes nearest
E.MULX.I.C.8.Z	Ensemble multiply extract immediate complex bytes zero
E.MULX.I.C.16.C	Ensemble multiply extract immediate complex doublets ceiling
E.MULX.I.C.16.F	Ensemble multiply extract immediate complex doublets floor
E.MULX.I.C.16.N	Ensemble multiply extract immediate complex doublets nearest
E.MULX.I.C.16.Z	Ensemble multiply extract immediate complex doublets zero
E.MULX.I.C.32.C	Ensemble multiply extract immediate complex quadlets ceiling
E.MULX.I.C.32.F	Ensemble multiply extract immediate complex quadlets floor
E.MULX.I.C.32.N	Ensemble multiply extract immediate complex quadlets nearest
E.MULX.I.C.32.Z	Ensemble multiply extract immediate complex quadlets zero
E.MULX.I.C.64.C	Ensemble multiply extract immediate complex octlets ceiling
E.MULX.I.C.64.F	Ensemble multiply extract immediate complex octlets floor
E.MULX.I.C.64.N	Ensemble multiply extract immediate complex octlets nearest
E.MULX.I.C.64.Z	Ensemble multiply extract immediate complex octlets zero
E.MULX.I.M.8.C	Ensemble multiply extract immediate mixed-signed bytes ceiling
E.MULX.I.M.8.F	Ensemble multiply extract immediate mixed-signed bytes floor
E.MULX.I.M.8.N	Ensemble multiply extract immediate mixed-signed bytes nearest
E.MULX.I.M.8.Z	Ensemble multiply extract immediate mixed-signed bytes zero
E.MULX.I.M.16.C	Ensemble multiply extract immediate mixed-signed doublets ceiling
E.MULX.I.M.16.F	Ensemble multiply extract immediate mixed-signed doublets floor
E.MULX.I.M.16.N	Ensemble multiply extract immediate mixed-signed doublets nearest
E.MULX.I.M.16.Z	Ensemble multiply extract immediate mixed-signed doublets zero
E.MULX.I.M.32.C	Ensemble multiply extract immediate mixed-signed quadlets ceiling
E.MULX.I.M.32.F	Ensemble multiply extract immediate mixed-signed quadlets floor
E.MULX.I.M.32.N	Ensemble multiply extract immediate mixed-signed quadlets nearest
E.MULX.I.M.32.Z	Ensemble multiply extract immediate mixed-signed quadlets zero
E.MULX.I.M.64.C	Ensemble multiply extract immediate mixed-signed octlets ceiling
E.MULX.I.M.64.F	Ensemble multiply extract immediate mixed-signed octlets floor
E.MULX.I.M.64.N	Ensemble multiply extract immediate mixed-signed octlets nearest
E.MULX.I.M.64.Z	Ensemble multiply extract immediate mixed-signed octlets zero
E.MULX.I.U.8.C	Ensemble multiply extract immediate unsigned bytes ceiling
E.MULX.I.U.8.F	Ensemble multiply extract immediate unsigned bytes floor
E.MULX.I.U.8.N	Ensemble multiply extract immediate unsigned bytes nearest
E.MULX.I.U.16.C	Ensemble multiply extract immediate unsigned doublets ceiling
E.MULX.I.U.16.F	Ensemble multiply extract immediate unsigned doublets floor
E.MULX.I.U.16.N	Ensemble multiply extract immediate unsigned doublets nearest
E.MULX.I.U.32.C	Ensemble multiply extract immediate unsigned quadlets ceiling
E.MULX.I.U.32.F	Ensemble multiply extract immediate unsigned quadlets floor

E.MULX.I.U.32.N	Ensemble multiply extract immediate unsigned quadlets nearest
E.MULX.I.U.64.C	Ensemble multiply extract immediate unsigned octlets ceiling
E.MULX.I.U.64.F	Ensemble multiply extract immediate unsigned octlets floor
E.MULX.I.U.64.N	Ensemble multiply extract immediate unsigned octlets nearest

Format

E.op.size.rnd rd=rc,rb,i

rd=eopsizernd(rc,rb,i)



```
sz ← log(size) - 3
case op of
    E.EXTRACT.I, E.EXTRACT.I.U, E.MULX.I, E.MULX.I.U, E.MULX.I.M:
        assert size ≥ i ≥ size-3
        sh ← size - i
    E.MULX.I.C:
        assert size+1 ≥ i ≥ size-2
        sh ← size + 1 - i
endcase
```

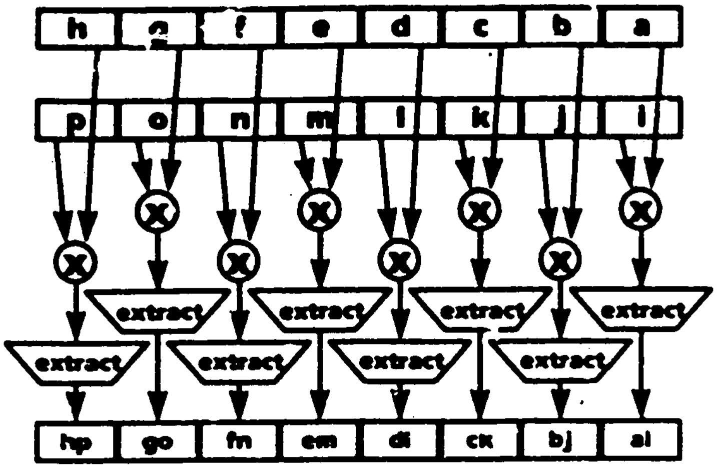
Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified and multiplied, added or subtracted, or are catenated and partitioned into operands of twice the size specified. The group of values is rounded, and limited as specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

For mixed-signed multiplies, the contents of register rc is signed, and the contents of register rb is unsigned. The extraction operation and the result of mixed-signed multiplies is signed.

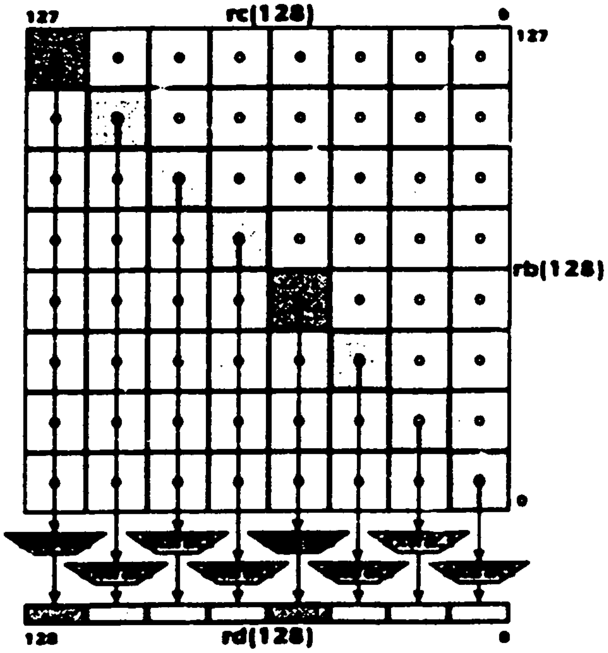
Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

An ensemble multiply extract immediate doublets instruction (E.MULXI.16 or E.MUL.X.I.U.16) multiplies operand [h g f e d c b a] by operand [p o n m l k j i], yielding the products [hp go fn em dl ck bj ai], rounded and limited as specified:



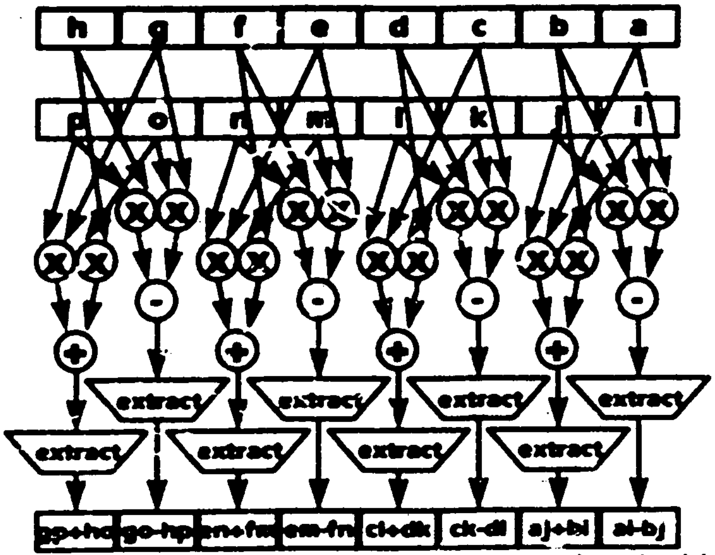
Ensemble multiply extract immediate doublets

Another illustration of ensemble multiply extract immediate doublets instruction (E.MUL.X.I.16 or E.MUL.X.I.U.16):



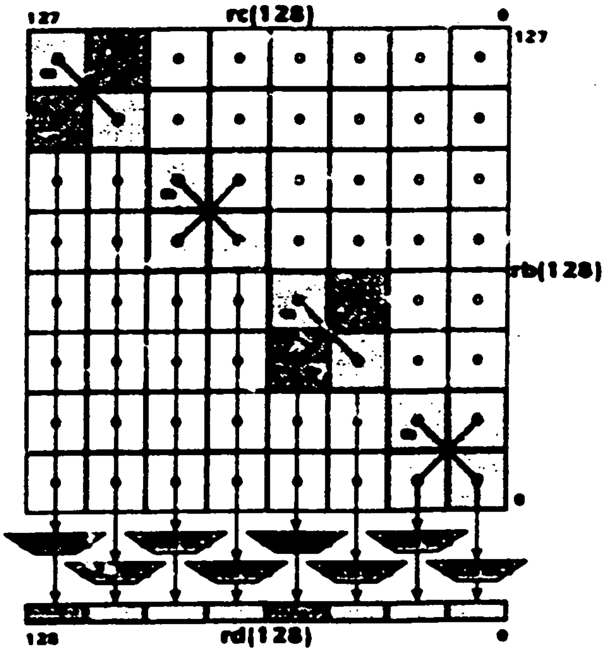
Ensemble multiply extract immediate doublets

An ensemble multiply extract immediate complex doublets instruction (E.MULXIC.16 or E.MUL.X.I.U.16) multiplies operand [h g f e d c b a] by operand [p o n m l k j i], yielding the result [gp+ho go-hp en+fm em-fn cl+dk ck-dl aj+bi ai-bj], rounded and limited as specified. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.:



Ensemble multiply extract immediate complex doublets

Another illustration of ensemble multiply extract immediate complex doublets instruction (E.MUL.X.I.C.16 or E.MUL.X.I.U.16):



Ensemble multiply extract immediate complex doublets

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← $((vs\&v_{size-1+i})^{h-size} \; || \; v_{size-1+i..i}) * ((ws\&w_{size-1+j})^{h-size} \; || \; w_{size-1+j..j})$
enddef

def EnsembleExtractImmediate(op,rnd,size,ra,rb,rc,sh)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.EXTRACT.I, E.MULX.I, F.MULX.I.C:
            as ← 1
            cs ← 1
            bs ← 1
        E.MULX.I.M:
            as ← 1
            cs ← 0
            bs ← 1
        E.EXTRACT.I.U, E.MULX.I.U:
            as ← 1
            cs ← 0
            bs ← 0
            if rnd = Z then
                raise ReservedInstruction
            endif
    endcase
    case op of
        E.EXTRACT.I, E.EXTRACT.I.U, E.MULX.I, E.MULX.I.U, E.MULX.I.M:
            h ← 2*size
        E.MULX.I.C:
            h ← (2*size) + 1
    endcase
    r ← h - size - sh
    for i ← 0 to 128-size by size
        case op of
            E.EXTRACT.I, E.EXTRACT.I.U:
                p ← $(c \; || \; b)_{2*(size+i)-1..2*i}$
            E.MULX.I, E.MULX.I.M, E.MULX.I.U:
                p ← mul(size,h,cs,c,i,bs,b,i)
            E.MULX.I.C:
                if i & size = 0 then
                    p ← mul(size,h,cs,c,i,bs,b,i) - mul(size,h,cs,c,i+size,bs,b,i+size)
                else
                    p ← mul(size,h,cs,c,i,bs,b,i+size) + mul(size,h,cs,c,i,bs,b,i+size)
                endif
        endcase
        case rnd of
            none, N:
                s ← $0^{h-r} \; || \; -p_r \; || \; p_r^{r-1}$
            Z:
                s ← $0^{h-r} \; || \; p^{r}_{h-1}$
            F:
                s ← $0^h$
            C:
                s ← $0^{h-r}$ || $1^r$
        endcase
        v ← ((as & $p_{h-1}$)||p) + (0||s)
        if ($v_{h..r+size}$ = (as & $v_{r+size-1}$)$^{h+1-r-size}$ then
            $a_{size-1+i..i}$ ← $v_{size-1+r..r}$
        else
            $a_{size-1+i..i}$ ← as ? ($v_h$ || ~$v_h^{size-1}$) : $1^{size}$
        endif
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

ReservedInstruction

Ensemble Extract Immediate Inplace

These operations take operands from two registers and a short immediate value, perform operations on partitions of bits in the operands, and place the catenated results in a third register.

Operation codes

E.MULADDX.I.C.8.C	Ensemble multiply add extract immediate signed complex bytes ceiling
E.MULADDX.I.C.8.F	Ensemble multiply add extract immediate signed complex bytes floor
E.MULADDX.I.C.8.N	Ensemble multiply add extract immediate signed complex bytes nearest
E.MULADDX.I.C.8.Z	Ensemble multiply add extract immediate signed complex bytes zero
E.MULADDX.I.C.16.C	Ensemble multiply add extract immediate signed complex doublets ceiling
E.MULADDX.I.C.16.F	Ensemble multiply add extract immediate signed complex doublets floor
E.MULADDX.I.C.16.N	Ensemble multiply add extract immediate signed complex doublets nearest
E.MULADDX.I.C.16.Z	Ensemble multiply add extract immediate signed complex doublets zero
E.MULADDX.I.C.32.C	Ensemble multiply add extract immediate signed complex quadlets ceiling
E.MULADDX.I.C.32.F	Ensemble multiply add extract immediate signed complex quadlets floor
E.MULADDX.I.C.32.N	Ensemble multiply add extract immediate signed complex quadlets nearest
E.MULADDX.I.C.32.Z	Ensemble multiply add extract immediate signed complex quadlets zero
E.MULADDX.I.C.64.C	Ensemble multiply add extract immediate signed complex octlets ceiling
E.MULADDX.I.C.64.F	Ensemble multiply add extract immediate signed complex octlets floor
E.MULADDX.I.C.64.N	Ensemble multiply add extract immediate signed complex octlets nearest
E.MULADDX.I.C.64.Z	Ensemble multiply add extract immediate signed complex octlets zero
E.MULADDX.I.M.8.C	Ensemble multiply add extract immediate mixed signed bytes ceiling
E.MULADDX.I.M.8.F	Ensemble multiply add extract immediate mixed signed bytes floor
E.MULADDX.I.M.8.N	Ensemble multiply add extract immediate mixed signed bytes nearest
E.MULADDX.I.M.8.Z	Ensemble multiply add extract immediate mixed signed bytes zero
E.MULADDX.I.M.16.C	Ensemble multiply add extract immediate mixed signed doublets ceiling
E.MULADDX.I.M.16.F	Ensemble multiply add extract immediate mixed signed doublets floor
E.MULADDX.I.M.16.N	Ensemble multiply add extract immediate mixed signed doublets nearest
E.MULADDX.I.M.16.Z	Ensemble multiply add extract immediate mixed signed doublets zero
E.MULADDX.I.M.32.C	Ensemble multiply add extract immediate mixed signed quadlets ceiling
E.MULADDX.I.M.32.F	Ensemble multiply add extract immediate mixed signed quadlets floor
E.MULADDX.I.M.32.N	Ensemble multiply add extract immediate mixed signed quadlets nearest
E.MULADDX.I.M.32.Z	Ensemble multiply add extract immediate mixed signed quadlets zero
E.MULADDX.I.M.64.C	Ensemble multiply add extract immediate mixed signed octlets ceiling
E.MULADDX.I.M.64.F	Ensemble multiply add extract immediate mixed signed octlets floor
E.MULADDX.I.M.64.N	Ensemble multiply add extract immediate mixed signed octlets nearest
E.MULADDX.I.M.64.Z	Ensemble multiply add extract immediate mixed signed octlets zero
E.MULADDX.I.8.C	Ensemble multiply add extract immediate signed bytes ceiling
E.MULADDX.I.8.F	Ensemble multiply add extract immediate signed bytes floor
E.MULADDX.I.8.N	Ensemble multiply add extract immediate signed bytes nearest
E.MULADDX.I.8.Z	Ensemble multiply add extract immediate signed bytes zero
E.MULADDX.I.16.C	Ensemble multiply add extract immediate signed doublets ceiling
E.MULADDX.I.16.F	Ensemble multiply add extract immediate signed doublets floor

E.MULADDX.I.16.N	Ensemble multiply add extract immediate signed doublets nearest
E.MULADDX.I.16.Z	Ensemble multiply add extract immediate signed doublets zero
E.MULADDX.I.32.C	Ensemble multiply add extract immediate signed quadlets ceiling
E.MULADDX.I.32.F	Ensemble multiply add extract immediate signed quadlets floor
E.MULADDX.I.32.N	Ensemble multiply add extract immediate signed quadlets nearest
E.MULADDX.I.32.Z	Ensemble multiply add extract immediate signed quadlets zero
E.MULADDX.I.64.C	Ensemble multiply add extract immediate signed octlets ceiling
E.MULADDX.I.64.F	Ensemble multiply add extract immediate signed octlets floor
E.MULADDX.I.64.N	Ensemble multiply add extract immediate signed octlets nearest
E.MULADDX.I.64.Z	Ensemble multiply add extract immediate signed octlets zero
E.MULADDX.I.U.8.C	Ensemble multiply add extract immediate unsigned bytes ceiling
E.MULADDX.I.U.8.F	Ensemble multiply add extract immediate unsigned bytes floor
E.MULADDX.I.U.8.N	Ensemble multiply add extract immediate unsigned bytes nearest
E.MULADDX.I.U.16.C	Ensemble multiply add extract immediate unsigned doublets ceiling
E.MULADDX.I.U.16.F	Ensemble multiply add extract immediate unsigned doublets floor
E.MULADDX.I.U.16.N	Ensemble multiply add extract immediate unsigned doublets nearest
E.MULADDX.I.U.32.C	Ensemble multiply add extract immediate unsigned quadlets ceiling
E.MULADDX.I.U.32.F	Ensemble multiply add extract immediate unsigned quadlets floor
E.MULADDX.I.U.32.N	Ensemble multiply add extract immediate unsigned quadlets nearest
E.MULADDX.I.U.64.C	Ensemble multiply add extract immediate unsigned octlets ceiling
E.MULADDX.I.U.64.F	Ensemble multiply add extract immediate unsigned octlets floor
E.MULADDX.I.U.64.N	Ensemble multiply add extract immediate unsigned octlets nearest

Format

E.op.size.rnd rd@rc,rb,i

rd=eopsizernd(rd,rc,rb,i)

31 24	23 18	17 12	11 6	5 4	3 2	1 0
E.op	rd	rc	rb	sz	rnd	sh
8	6	6	6	2	2	2

```
sz ← log(size) - 3
case op of
    E.MULADDX.I:
        sh ← size - i - 1
    E.MULADDX.I.U, E.MULADDX.I.M, E.MULADDX.I.C:
        sh ← size - i
endcase
```

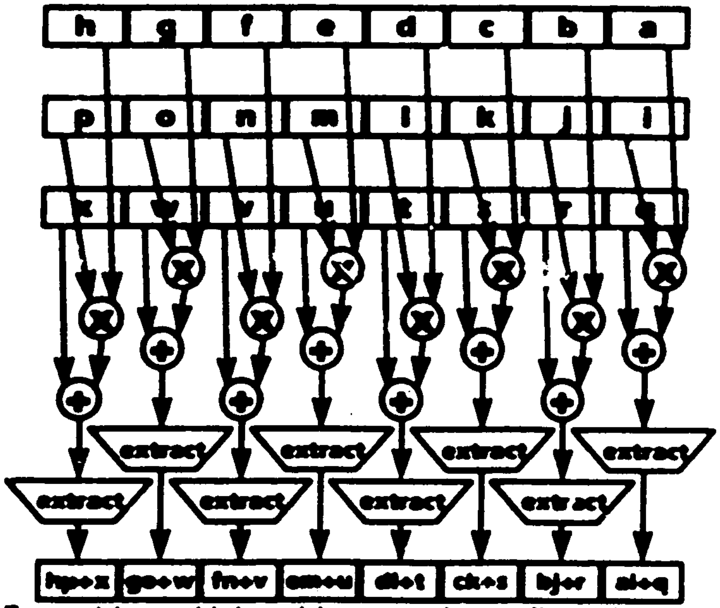
Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified and multiplied, added or subtracted, or are catenated and partitioned into operands of twice the size specified. The contents of register rd are partitioned into groups of operands of the size specified and sign or zero ensemble and shifted as specified then added to the group of values computed. The group of values is rounded, and limited as specified, yielding a group of results which is the size specified. The group of results is catenated and placed in register rd.

For mixed-signed multiplies, the contents of register rc is signed, and the contents of register rb as unsigned. The extraction operation, the contents of register rd, and the result of mixed-signed multiplies are signed.

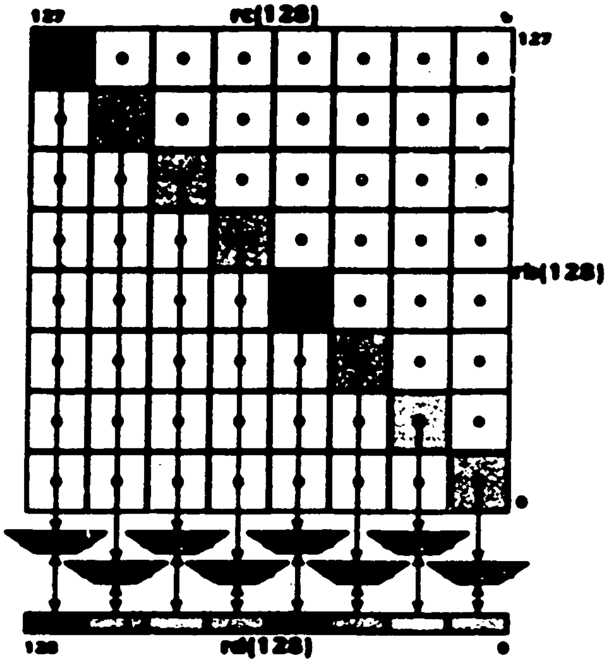
Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

An ensemble multiply add extract immediate doublets instruction (E.MULADD.X.I.16 or E.MULADD.X.I.U.16) multiplies operand [h g f e d c b a] by operand [p o n m l k j i], then adding [x w v u t s r q], yielding the products [hp+x go+w fn+v em+u dl+t ck+s bj+r ai+q], rounded and limited as specified:



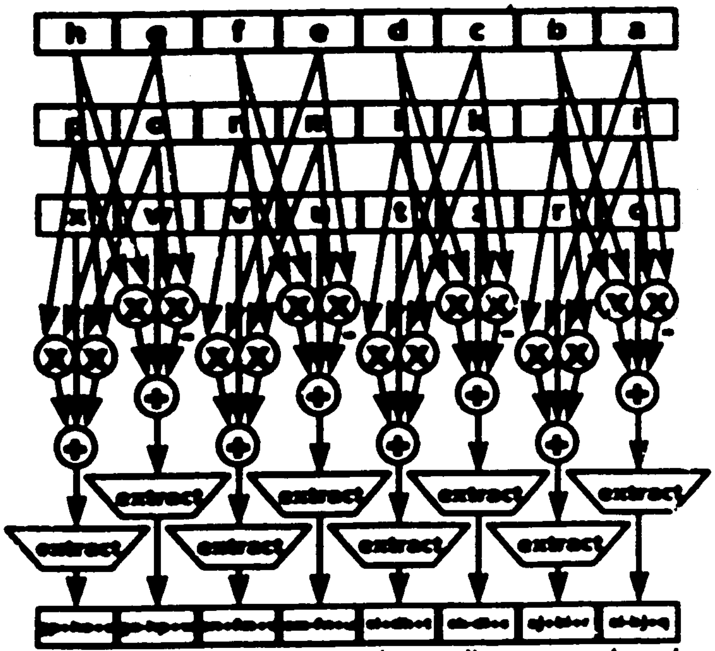
Ensemble multiply add extract immediate doublets

Another illustration of ensemble multiply add extract immediate doublets instruction (E.MULADDXI.16 or E.MUL.ADD.X.I.U.16):



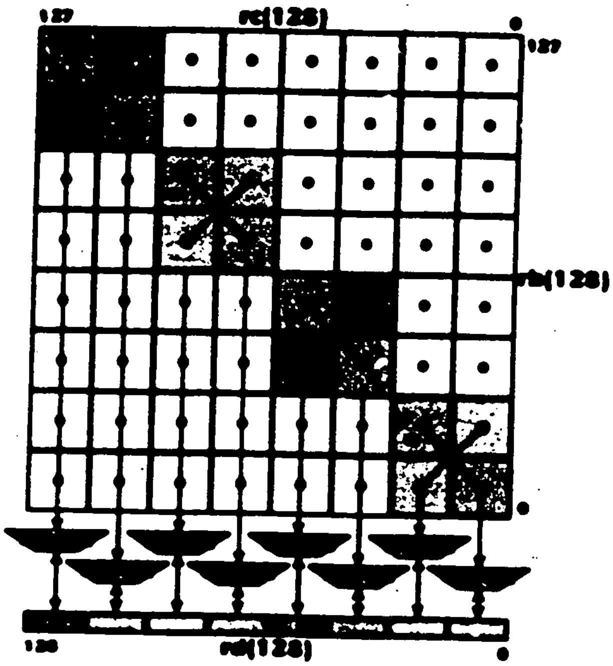
Ensemble multiply add extract immediate doublets

An ensemble multiply add extract immediate complex doublets instruction (E.MUL.ADD.X.I.C.16 or G.MUL.ADD.X.I.U.16) multiplies operand [h g f e d c b a] by operand [p o n m l k j i], then adding [x w v u t s r q], yielding the result [gp+ho+x go-hp+w en+fm+v em-fn+u cl+dk+t ck-dl+s aj+bi+r ai-bj+q], rounded and limited as specified. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.:



Ensemble multiply add extract immediate complex doublets

Another illustration of ensemble multiply extract immediate complex doublets instruction (E.MUL.ADD.X.I.C.16).:



Ensemble multiply add extract immediate complex doublets

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def EnsembleExtractImmediateInplace(op,rnd,size,rd,rc,rb,sh)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MUL.ADD.X.I, E.MUL.ADD.X.I.C:
            ds ← 1
            cs ← 1
            bs ← 1
        E.MUL.ADD.X.I.M:
            ds ← 1
            cs ← 0
            bs ← 1
        E.MUL.ADD.X.I.U:
            ds ← 0
            cs ← 0
            bs ← 0
            if rnd = Z then
                raise ReservedInstruction
            endif
    endcase
    case op of
        E.MULADD.X.I, E.MULADD.X.I.U, E.MULADD.X.I.M:
            h ← 2*size + 1
        E.MULADD.X.I.C:
            h ← (2*size) + 2
    endcase
    r ← h - size - sh - 1 - (cs and bs)
    for i ← 0 to 128-size by size
        di ← $((ds \text{ and } d_{i+size-1})^{h-size-r} \,||\, (d_{i+size-1..i} \,||\, 0^r)$
        case op of
            E.MULADD.X.I, E.MULADD.X.I.M, E.MULADD.X.I.U:
                p ← mul(size,h,cs,c,i,bs,b,i) + di
            E.MULADD.X.I.C:
                if i & size = 0 then
                    p ← mul(size,h,cs,c,i,bs,b,i) - mul(size,h,cs,c,i+size,bs,b,i+size) + di
                else
                    p ← mul(size,h,cs,c,i,bs,b,i+size) + mul(size,h,cs,c,i,bs,b,i+size) + di
                endif
        endcase
        case rnd of
            none, N:
                s ← $0^{h-r} \,||\, \sim p_r \,||\, p_r^{r-1}$
            Z:
                s ← $0^{h-r} \,||\, p_{h-1}^r$
            F:
                s ← $0^h$
            C:
                s ← $0^{h-r} \,||\, 1^r$
        endcase
        v ← $((ds \,\&\, p_{h-1}) \,||\, p) + (0 \,||\, s)$
        if $(v_{h..r+size} = (ds \,\&\, v_{r+size-1})^{h+1-r-size}$ then
                $a_{size-1+i..i}$ ← $v_{size-1+r..r}$
        else
                $a_{size-1+i..i}$ ← ds ? $(v_h \,||\, \sim v_h^{size-1}) : 1^{size}$
        endif
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

ReservedInstruction

Ensemble Floating-point

These operations take two values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the catenated results in a register.

Operation codes

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double
E.DIV.F.64.C	Ensemble divide floating-point double ceiling

E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

Selection

class	op	prec				round/trap					
add	EADDF	16	32	64	128	NONE	C	F	N	X	Z
divide	EDIVF	16	32	64	128	NONE	C	F	N	X	Z
multiply	EMULF	16	32	64	128	NONE	C	F	N	X	Z
complex multiply	EMUL.CF	16	32	64		NONE					

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	rb	op.round
8	6	6	6	6

Description

The contents of registers ra and rb are combined using the specified floating-point operation The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,$v_{size-1+i..i}$),F(size,$w_{size-1+j..j}$))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        $c_i$ ← F(prec,$c_{i+prec-1..i}$)
        $b_i$ ← F(prec,$b_{i+prec-1..i}$)
        case op of
            E.ADD.F:
                $a_i$ ← faddr($c_i$,$b_i$,round)
            E.MUL.F:
                $a_i$ ← fmul($c_i$,$b_i$)
            E.MUL.C.F:
                if (i and prec) then
                    $a_i$ ← fadd(mul(prec,c,i,b,i-prec), mul(prec,c,i-prec,b,i))
                else
                    $a_i$ ← fsub(mul(prec,c,I,b,I), mul(prec,c,i+prec,b,i+prec))
                endif
            E.DIV.F:
                $a_i$ ← fdiv($c_i$,$b_i$)
        endcase
        $a_{i+prec-1..i}$ ← PackF(prec, $a_i$, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating point arithmetic

Ensemble Inplace

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in the third register.

Operation codes

E.MULADD.8	Ensemble multiply signed bytes add doublets
E.MULADD.16	Ensemble multiply signed doublets add quadlets
E.MULADD.32	Ensemble multiply signed quadlets add octlets
E.MULADD.64	Ensemble multiply signed octlets add hexlet
E.MULADD.C.8	Ensemble multiply complex bytes add doublets
E.MULADD.C.16	Ensemble multiply complex doublets add quadlets
E.MULADD.C.32	Ensemble multiply complex quadlets add octlets
E.MULADD.M.8	Ensemble multiply mixed-signed bytes add doublets
E.MULADD.M.16	Ensemble multiply mixed-signed doublets add quadlets
E.MULADD.M.32	Ensemble multiply mixed-signed quadlets add octlets
E.MULADD.M.64	Ensemble multiply mixed-signed octlets add hexlet
E.MULADD.U.8	Ensemble multiply unsigned bytes add doublets
E.MULADD.U.16	Ensemble multiply unsigned doublets add quadlets
E.MULADD.U.32	Ensemble multiply unsigned quadlets add octlets
E.MULADD.U.64	Ensemble multiply unsigned octlets add hexlet
E.MUL.SUB.8	Ensemble multiply signed bytes subtract doublets
E.MUL.SUB.16	Ensemble multiply signed doublets subtract quadlets
E.MUL.SUB.32	Ensemble multiply signed quadlets subtract octlets
E.MUL.SUB.64	Ensemble multiply signed octlets subtract hexlet
E.MUL.SUB.C.8	Ensemble multiply complex bytes subtract doublets
E.MUL.SUB.C.16	Ensemble multiply complex doublets subtract quadlets
E.MUL.SUB.C.32	Ensemble multiply complex quadlets subtract octlets
E.MUL.SUB.M.8	Ensemble multiply mixed-signed bytes subtract doublets
E.MUL.SUB.M.16	Ensemble multiply mixed-signed doublets subtract quadlets
E.MUL.SUB.M.32	Ensemble multiply mixed-signed quadlets subtract octlets
E.MUL.SUB.M.64	Ensemble multiply mixed-signed octlets subtract hexlet
E.MUL.SUB.U.8	Ensemble multiply unsigned bytes subtract doublets
E.MUL.SUB.U.16	Ensemble multiply unsigned doublets subtract quadlets
E.MUL.SUB.U.32	Ensemble multiply unsigned quadlets subtract octlets
E.MUL.SUB.U.64	Ensemble multiply unsigned octlets subtract hexlet

Selection

class	op	type	prec
multiply	E.MULADD	NONE M U	8 16 32 64
complex multiply	E.MUL.SUB	C	8 16 32

Format

E.op.size rd=rc,rb

rd=gopsize(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
E.size	rd	rc	rb	op
8	6	6	6	6

Description

The contents of registers rd, rc and rb are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Register rd is both a source and destination of this instruction.

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← $((vs\&v_{size-1+i})^{h-size} \,||\, v_{size-1+i..i}) * ((ws\&w_{size-1+j})^{h-size} \,||\, w_{size-1+j..j})$
enddef

def EnsembleInplace(op,size,rd,rc,rb) as
    if size=1 then
        raise ReservedInstruction
    endif
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MULADD, E.MULSUB, E.MULADDC, E.MULSUBC:
            cs ← 1
            bs ← 1
        E.MULADDM, E.MULSUBM:
            cs ← 0
            bs ← 1
        E.MULADDU, E.MULSUBU:
            cs ← 0
            bs ← 0
    endcase
    h ← 2*size
    for i ← 0 to 64-size by size
        di ← $d_{2*(i+size)-1..2*i}$
        case op of
            E.MULADD, E.MULADDU, E.MULADDM:
                p ← mul(size,h,cs,c,i,bs,b,i) + di
            E.MULADDC:
                if i & size = 0 then
                    p ← mul(size,h,cs,c,i,bs,b,i) - mul(size,h,cs,c,i+size,bs,b,i+size) + di
                else
                    p ← mul(size,h,cs,c,i,bs,b,i+size) + mul(size,h,cs,c,i,bs,b,i+size) + di
                endif
            E.MULSUB, E.MULSUB.U, E.MULSUB.M:
                p ← mul(size,h,cs,c,i,bs,b,i) - di
            E.MULSUBC:
                if i & size = 0 then
                    p ← mul(size,h,cs,c,i,bs,b,i) - mul(size,h,cs,c,i+size,bs,b,i+size) - di
                else
                    p ← mul(size,h,cs,c,i,bs,b,i+size) + mul(size,h,cs,c,i,bs,b,i+size) - di
                endif
        endcase
        $a_{2*(i+size)-1..2*i}$ ← p
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Inplace Floating-point

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in the third register.

Operation codes

E.MULADD.C.F.16	Ensemble multiply add complex floating-point half
E.MULADD.C.F.32	Ensemble multiply add complex floating-point single
E.MULADD.C.F.64	Ensemble multiply add complex floating-point double
E.MULADD.F.16	Ensemble multiply add floating-point half
E.MULADD.F.16.C	Ensemble multiply add floating-point half ceiling
E.MULADD.F.16.F	Ensemble multiply add floating-point half floor
E.MULADD.F.16.N	Ensemble multiply add floating-point half nearest
E.MULADD.F.16.X	Ensemble multiply add floating-point half exact
E.MULADD.F.16.Z	Ensemble multiply add floating-point half zero
E.MULADD.F.32	Ensemble multiply add floating-point single
E.MULADD.F.32.C	Ensemble multiply add floating-point single ceiling
E.MULADD.F.32.F	Ensemble multiply add floating-point single floor
E.MULADD.F.32.N	Ensemble multiply add floating-point single nearest
E.MULADD.F.32.X	Ensemble multiply add floating-point single exact
E.MULADD.F.32.Z	Ensemble multiply add floating-point single zero
E.MULADD.F.64	Ensemble multiply add floating-point double
E.MULADD.F.64.C	Ensemble multiply add floating-point double ceiling
E.MULADD.F.64.F	Ensemble multiply add floating-point double floor
E.MULADD.F.64.N	Ensemble multiply add floating-point double nearest
E.MULADD.F.64.X	Ensemble multiply add floating-point double exact
E.MULADD.F.64.Z	Ensemble multiply add floating-point double zero
E.MULADD.F.128	Ensemble multiply add floating-point quad
E.MULADD.F.128.C	Ensemble multiply add floating-point quad ceiling
E.MULADD.F.128.F	Ensemble multiply add floating-point quad floor
E.MULADD.F.128.N	Ensemble multiply add floating-point quad nearest
E.MULADD.F.128.X	Ensemble multiply add floating-point quad exact
E.MULADD.F.128.Z	Ensemble multiply add floating-point quad zero
E.MUL.SUB.C.F.16	Ensemble multiply subtract complex floating-point half
E.MUL.SUB.C.F.32	Ensemble multiply subtract complex floating-point single
E.MUL.SUB.C.F.64	Ensemble multiply subtract complex floating-point double
E.MUL.SUB.F.16	Ensemble multiply subtract floating-point half
E.MUL.SUB.F.32	Ensemble multiply subtract floating-point single
E.MUL.SUB.F.64	Ensemble multiply subtract floating-point double
E.MUL.SUB.F.128	Ensemble multiply subtract floating-point quad

Selection

class	op	type	prec	round/trap
multiply add	E.MULADD	F	16 32 64 128	NONE C F N X Z
		C.F	16 32 64	NONE
multiply subtract	E.MULSUB	F	16 32 64 128	NONE
		C.F	16 32 64	NONE

Format

E.op.size rd@rc,rb

rd=eopsize(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
E.size	rd	rc	rb	op
8	6	6	6	6

Description

The contents of registers rd, rc and rb are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Register rd is both a source and destination of this instruction.

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,$v_{size-1+i..i}$),F(size,$w_{size-1+j..j}$))
enddef

oef EnsembleInplaceFloatingPoint(op,size,rd,rc,rb) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        di ← F(prec,$d_{i+prec-1..i}$)
        case op of
            E.MULADD.F:
                ai ← fadd(di, mul(prec,c,i,b,i))
            E.MULADD.C.F:
                if (i and prec) then
                    ai ← fadd(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← fadd(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
            E.MUL.SUB.F:
                ai ← frsub(di, mul(prec,c,i,b,i))
            E.MUL.SUB.C.F:
                if (i and prec) then
                    ai ← frsub(di, fadd(mul(prec,c,i,b,i-prec), mul(c,i-prec,b,i)))
                else
                    ai ← frsub(di, fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec)))
                endif
            endcase
            $a_{i+prec-1..i}$ ← PackF(prec, ai, round)
        endfor
        RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Reversed Floating-point

These operations take two values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

Selection

class	op	prec				round/trap
set	SET. E LG L GE	16	32	64	128	NONE X
subtract	SUB	16	32	64	128	NONE C F N X Z

Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	rb	op.round
8	6	6	6	6

Description

The contents of registers rc and rb are combined using the specified floating-point operation. The result is placed in register rd. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 129)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,$c_{i+prec-1..i}$)
        bi ← F(prec,$b_{i+prec-1..i}$)
        ai ← frsubr(ci,-bi, round)
        $a_{i+prec-1..i}$ ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating point arithmetic

Ensemble Ternary

These operations take three values from registers, perform a group of calculations on partitions of bits of the operands and place the catenated results in a fourth register.

Operation codes

E.MUL.G.8	*Ensemble multiply Galois field byte*
E.MUL.G.64	*Ensemble multiply Galois field octlet*

Format

E.MUL.G.size ra=rd,rc,rb

ra=emulgsize(rd,rc,rb)

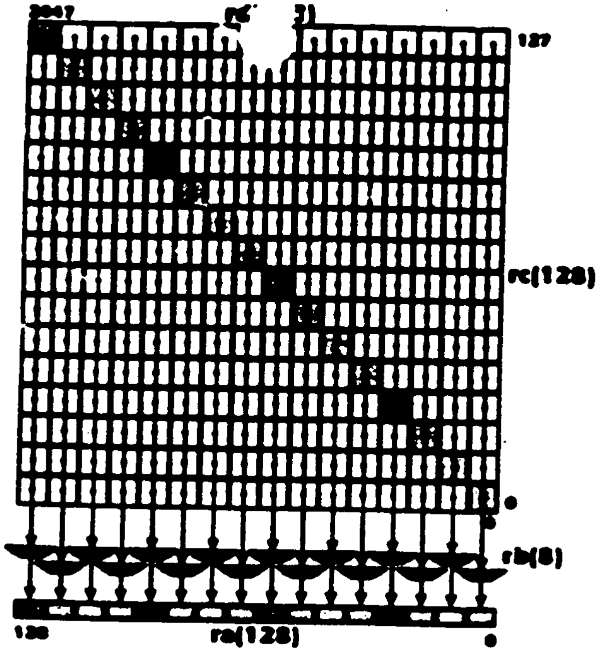
31 24	23 18	17 12	11 6	5 0
E.MUL.G.size	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

The contents of registers rd and rc are partitioned into groups of operands of the size specified and multiplied in the manner of polynomials. The group of values is reduced modulo the polynomial specified by the contents of register rb, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register ra.

An ensemble multiply Galois field bytes instruction (E.MULG.8) multiplies operand [d15 d14 d13 d12 d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0] by operand [c15 c14 c13 c12 c11 c10 c9 c8 c7 c6 c5 c4 c3 c2 c1 c0], modulo polynomial [q], yielding the results [(d15c15 mod q) (d14c14 mod q) … (d0c0 mod q):



Ensemble multiply Galois field bytes

Definition

```
def c ← PolyMultiply(size,a,b) as
    p[0] ← $0^{2*size}$
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ $a_k$ ? ($0^{size-k}$ || b || $0^k$) : $0^{2*size}$
    endfor
    c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
    p[0] ← a
    for k ← size-1 to 0 by -1
        p[k+1] ← p[k] ^ $p[0]_{size+k}$ ? ($0^{size-k}$ || $1^1$ || b || $0^k$) : $0^{2*size}$
    endfor
    c ← $p[size]_{size-1..0}$
enddef

def EnsembleTernary(op,size,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
```

```
        E.MUL.G:
            for i ← 0 to 128-size by size
                $a_{size-1+i..i}$ ← PolyResidue(size,PolyMul(size,$c_{size-1+i..i}$,$b_{size-1+i..i}$),$d_{size-1+i..i}$)
            endfor
    endcase
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Ensemble Ternary Floating-point

These operations take three values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

E.SCALADD.F.16	Ensemble scale add floating-point half
E.SCALADD.F.32	Ensemble scale add floating-point single
E.SCALADD.F.64	Ensemble scale add floating-point double

Selection

class	op	prec
scale add	E.SCALADD.F	16 32 64

Format

E.SCALADD.F.size ra=rd,rc,rb

ra=escaladdfsize(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd and rc are taken to represent a group of floating-point operands. Operands from register rd are multiplied with a floating-point operand taken from the least-significant bits of the contents of register rb and added to operands from register rc multiplied with a floating-point operand taken from the next least-significant bits of the contents of register rb. The results are concatenated and placed in register ra. The results are rounded to the nearest representable floating-point value in a single floating-point operation. Floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. These instructions cannot select a directed rounding mode or trap on inexact.

Definition

```
def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        di ← F(prec,$d_{i+prec-1..i}$)
        ci ← F(prec,$c_{i+prec-1..i}$)
        ai ← fadd(fmul(di, F(prec,$b_{prec-1..0}$)), fmul(ci, F(prec,$b_{2*prec-1..prec}$)))
        $a_{i+prec-1..i}$ ← PackF(prec, ai, none)
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Ensemble Unary

These operations take operands from a register, perform operations on partitions of bits in the operand, and place the concatenated results in a second register.

Operation codes

E.LOG.MOST.8	Ensemble log of most significant bit signed bytes
E.LOG.MOST.16	Ensemble log of most significant bit signed doublets
E.LOG.MOST.32	Ensemble log of most significant bit signed quadlets
E.LOG.MOST.64	Ensemble log of most significant bit signed octlets
E.LOG.MOST.128	Ensemble log of most significant bit signed hexlet
E.LOG.MOST.U.8	Ensemble log of most significant bit unsigned bytes
E.LOG.MOST.U.16	Ensemble log of most significant bit unsigned doublets
E.LOG.MOST.U.32	Ensemble log of most significant bit unsigned quadlets
E.LOG.MOST.U.64	Ensemble log of most significant bit unsigned octlets
E.LOG.MOST.U.128	Ensemble log of most significant bit unsigned hexlet
E.SUM.8	Ensemble sum signed bytes
E.SUM.16	Ensemble sum signed doublets
E.SUM.32	Ensemble sum signed quadlets
E.SUM.64	Ensemble sum signed octlets
E.SUM.U.1[26]	Ensemble sum unsigned bits
E.SUM.U.8	Ensemble sum unsigned bytes
E.SUM.U.16	Ensemble sum unsigned doublets
E.SUM.U.32	Ensemble sum unsigned quadlets
E.SUM.U.64	Ensemble sum unsigned octlets

Selection

class	op		size
sum	SUM		8 16 32 64
	SUM.U		1 8 16 32 64
log most significant bit	LOG.MOST	LOG.MOST.U	8 16 32 64 128

Format

E.op.size rd=rc

rd=eopsize(rc)

31 24	23 18	17 12	11 6	5 0
E.size	rd	rc	op	E.UNARY
8	6	6	6	6

[26] E.SUM.U.1 is encoded as E.SUM.U.128.

Description

Values are taken from the contents of register rc. The specified operation is performed, and the result is placed in register rd.

Definition

```
def EnsembleUnary(op,size,rd,rc)
    c ← RegRead(rc, 128)
    case op of
        E.LOG.MOST:
            for i ← 0 to 128-size by size
                if ($c_{i+size-1..i}$ = 0) then
                    $a_{i+size-1..i}$ ← -1
                else
                    for j ← 0 to size-1
                        if $c_{size-1+i..j+i}$ = ($c_{size-1+i}^{size-1-j}$ || not $c_{size-1+i}$) then
                            $a_{i+size-1..i}$ ← j
                        endif
                    endfor
                endif
            endfor
        E.LOG.MOSTU:
            for i ← 0 to 128-size by size
                if ($c_{i+size-1..i}$ = 0) then
                    $a_{i+size-1..i}$ ← -1
                else
                    for j ← 0 to size-1
                        if $c_{size-1+i..j+i}$ = ($0^{size-1-j}$ || 1) then
                            $a_{i+size-1..i}$ ← j
                        endif
                    endfor
                endif
            endfor
        E.SUM:
            p[0] ← $0^{128}$
            for i ← 0 to 128-size by size
                p[i+size] ← p[i] + ($c_{size-1+i}^{128-size}$ || $c_{size-1+i..i}$)
            endfor
            a ← p[128]
        E.SUMU:
            p[0] ← $0^{128}$
            for i ← 0 to 128-size by size
                p[i+size] ← p[i] + ($0^{128-size}$ || $c_{size-1+i..i}$)
            endfor
            a ← p[128]
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Unary Floating-point

These operations take one value from a register, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

E.ABS.F.16	Ensemble absolute value floating-point half
E.ABS.F.16.X	Ensemble absolute value floating-point half exception
E.ABS.F.32	Ensemble absolute value floating-point single
E.ABS.F.32.X	Ensemble absolute value floating-point single exception
E.ABS.F.64	Ensemble absolute value floating-point double
E.ABS.F.64.X	Ensemble absolute value floating-point double exception
E.ABS.F.128	Ensemble absolute value floating-point quad
E.ABS.F.128.X	Ensemble absolute value floating-point quad exception
E.COPY.F.16	Ensemble copy floating-point half
E.COPY.F.16.X	Ensemble copy floating-point half exception
E.COPY.F.32	Ensemble copy floating-point single
E.COPY.F.32.X	Ensemble copy floating-point single exception
E.COPY.F.64	Ensemble copy floating-point double
E.COPY.F.64.X	Ensemble copy floating-point double exception
E.COPY.F.128	Ensemble copy floating-point quad
E.COPY.F.128.X	Ensemble copy floating-point quad exception
E.DEFLATE.F.32	Ensemble convert floating-point half from single
E.DEFLATE.F.32.C	Ensemble convert floating-point half from single ceiling
E.DEFLATE.F.32.F	Ensemble convert floating-point half from single floor
E.DEFLATE.F.32.N	Ensemble convert floating-point half from single nearest
E.DEFLATE.F.32.X	Ensemble convert floating-point half from single exact
E.DEFLATE.F.32.Z	Ensemble convert floating-point half from single zero
E.DEFLATE.F.64	Ensemble convert floating-point single from double
E.DEFLATE.F.64.C	Ensemble convert floating-point single from double ceiling
E.DEFLATE.F.64.F	Ensemble convert floating-point single from double floor
E.DEFLATE.F.64.N	Ensemble convert floating-point single from double nearest
E.DEFLATE.F.64.X	Ensemble convert floating-point single from double exact
E.DEFLATE.F.64.Z	Ensemble convert floating-point single from double zero
E.DEFLATE.F.128	Ensemble convert floating-point double from quad
E.DEFLATE.F.128.C	Ensemble convert floating-point double from quad ceiling
E.DEFLATE.F.128.F	Ensemble convert floating-point double from quad floor
E.DEFLATE.F.128.N	Ensemble convert floating-point double from quad nearest
E.DEFLATE.F.128.X	Ensemble convert floating-point double from quad exact
E.DEFLATE.F.128.Z	Ensemble convert floating-point double from quad zero
E.FLOAT.F.16	Ensemble convert floating-point half from doublets
E.FLOAT.F.16.C	Ensemble convert floating-point half from doublets ceiling
E.FLOAT.F.16.F	Ensemble convert floating-point half from doublets floor
E.FLOAT.F.16.N	Ensemble convert floating-point half from doublets nearest

E.FLOAT.F.16.X	Ensemble convert floating-point half from doublets exact
E.FLOAT.F.16.Z	Ensemble convert floating-point half from doublets zero
E.FLOAT.F.32	Ensemble convert floating-point single from quadlets
E.FLOAT.F.32.C	Ensemble convert floating-point single from quadlets ceiling
E.FLOAT.F.32.F	Ensemble convert floating-point single from quadlets floor
E.FLOAT.F.32.N	Ensemble convert floating-point single from quadlets nearest
E.FLOAT.F.32.X	Ensemble convert floating-point single from quadlets exact
E.FLOAT.F.32.Z	Ensemble convert floating-point single from quadlets zero
E.FLOAT.F.64	Ensemble convert floating-point double from octlets
E.FLOAT.F.64.C	Ensemble convert floating-point double from octlets ceiling
E.FLOAT.F.64.F	Ensemble convert floating-point double from octlets floor
E.FLOAT.F.64.N	Ensemble convert floating-point double from octlets nearest
E.FLOAT.F.64.X	Ensemble convert floating-point double from octlets exact
E.FLOAT.F.64.Z	Ensemble convert floating-point double from octlets zero
E.FLOAT.F.128	Ensemble convert floating-point quad from hexlet
E.FLOAT.F.128.C	Ensemble convert floating-point quad from hexlet ceiling
E.FLOAT.F.128.F	Ensemble convert floating-point quad from hexlet floor
E.FLOAT.F.128.N	Ensemble convert floating-point quad from hexlet nearest
E.FLOAT.F.128.X	Ensemble convert floating-point quad from hexlet exact
E.FLOAT.F.128.Z	Ensemble convert floating-point quad from hexlet zero
E.INFLATE.F.16	Ensemble convert floating-point single from half
E.INFLATE.F.16.X	Ensemble convert floating-point single from half exception
E.INFLATE.F.32	Ensemble convert floating-point double from single
E.INFLATE.F.32.X	Ensemble convert floating-point double from single exception
E.INFLATE.F.64	Ensemble convert floating-point quad from double
E.INFLATE.F.64.X	Ensemble convert floating-point quad from double exception
E.NEG.F.16	Ensemble negate floating-point half
E.NEG.F.16.X	Ensemble negate floating-point half exception
E.NEG.F.32	Ensemble negate floating-point single
E.NEG.F.32.X	Ensemble negate floating-point single exception
E.NEG.F.64	Ensemble negate floating-point double
E.NEG.F.64.X	Ensemble negate floating-point double exception
E.NEG.F.128	Ensemble negate floating-point quad
E.NEG.F.128.X	Ensemble negate floating-point quad exception
E.RECEST.F.16	Ensemble reciprocal estimate floating-point half
E.RECEST.F.16.X	Ensemble reciprocal estimate floating-point half exception
E.RECEST.F.32	Ensemble reciprocal estimate floating-point single
E.RECEST.F.32.X	Ensemble reciprocal estimate floating-point single exception
E.RECEST.F.64	Ensemble reciprocal estimate floating-point double
E.RECEST.F.64.X	Ensemble reciprocal estimate floating-point double exception
E.RECEST.F.128	Ensemble reciprocal estimate floating-point quad
E.RECEST.F.128.X	Ensemble reciprocal estimate floating-point quad exception
E.RSQREST.F.16	Ensemble floating-point reciprocal square root estimate half
E.RSQREST.F.16.X	Ensemble floating-point reciprocal square root estimate half exact
E.RSQREST.F.32	Ensemble floating-point reciprocal square root estimate single
E.RSQREST.F.32.X	Ensemble floating-point reciprocal square root estimate single exact

E.RSQREST.F.64	Ensemble floating-point reciprocal square root estimate double
E.RSQREST.F.64.X	Ensemble floating-point reciprocal square root estimate double exact
E.RSQREST.F.128	Ensemble floating-point reciprocal square root estimate quad
E.RSQREST.F.128.X	Ensemble floating-point reciprocal square root estimate quad exact
E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16.C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default
E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
F.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default
E.SQR.F.16	Ensemble square root floating-point half
E.SQR.F.16.C	Ensemble square root floating-point half ceiling
E.SQR.F.16.F	Ensemble square root floating-point half floor
E.SQR.F.16.N	Ensemble square root floating-point half nearest
E.SQR.F.16.X	Ensemble square root floating-point half exact
E.SQR.F.16.Z	Ensemble square root floating-point half zero

E.SQR.F.32	Ensemble square root floating-point single
E.SQR.F.32.C	Ensemble square root floating-point single ceiling
E.SQR.F.32.F	Ensemble square root floating-point single floor
E.SQR.F.32.N	Ensemble square root floating-point single nearest
E.SQR.F.32.X	Ensemble square root floating-point single exact
E.SQR.F.32.Z	Ensemble square root floating-point single zero
E.SQR.F.64	Ensemble square root floating-point double
E.SQR.F.64.C	Ensemble square root floating-point double ceiling
E.SQR.F.64.F	Ensemble square root floating-point double floor
E.SQR.F.64.N	Ensemble square root floating-point double nearest
E.SQR.F.64.X	Ensemble square root floating-point double exact
E.SQR.F.64.Z	Ensemble square root floating-point double zero
E.SQR.F.128	Ensemble square root floating-point quad
E.SQR.F.128.C	Ensemble square root floating-point quad ceiling
E.SQR.F.128.F	Ensemble square root floating-point quad floor
E.SQR.F.128.N	Ensemble square root floating-point quad nearest
E.SQR.F.128.X	Ensemble square root floating-point quad exact
E.SQR.F.128.Z	Ensemble square root floating-point quad zero
E.SUM.F.16	Ensemble sum floating-point half
E.SUM.F.16.C	Ensemble sum floating-point half ceiling
E.SUM.F.16.F	Ensemble sum floating-point half floor
E.SUM.F.16.N	Ensemble sum floating-point half nearest
E.SUM.F.16.X	Ensemble sum floating-point half exact
E.SUM.F.16.Z	Ensemble sum floating-point half zero
E.SUM.F.32	Ensemble sum floating-point single
E.SUM.F.32.C	Ensemble sum floating-point single ceiling
E.SUM.F.32.F	Ensemble sum floating-point single floor
E.SUM.F.32.N	Ensemble sum floating-point single nearest
E.SUM.F.32.X	Ensemble sum floating-point single exact
E.SUM.F.32.Z	Ensemble sum floating-point single zero
E.SUM.F.64	Ensemble sum floating-point double
E.SUM.F.64.C	Ensemble sum floating-point double ceiling
E.SUM.F.64.F	Ensemble sum floating-point double floor
E.SUM.F.64.N	Ensemble sum floating-point double nearest
E.SUM.F.64.X	Ensemble sum floating-point double exact
E.SUM.F.64.Z	Ensemble sum floating-point double zero
E.SUM.F.128	Ensemble sum floating-point quad
E.SUM.F.128.C	Ensemble sum floating-point quad ceiling
E.SUM.F.128.F	Ensemble sum floating-point quad floor
E.SUM.F.128.N	Ensemble sum floating-point quad nearest
E.SUM.F.128.X	Ensemble sum floating-point quad exact
E.SUM.F.128.Z	Ensemble sum floating-point quad zero

Selection

	op	prec				round/trap
copy	COPY	16	32	64	128	NONE X
absolute valuc	ABS	16	32	64	128	NONE X
float from integer	FLOAT	16	32	64	128	NONE C F N X Z
integer from float	SINK	16	32	64	128	NONE C F N X Z C.D F.D Z.D
increase format precision	INFLATE	16	32	64		NONE X
decrease format precision	DEFLATE		32	64	128	NONE C F N X Z
negate	NEG	16	32	64	128	NONE X
reciprocal estimate	RECEST	16	32	64	128	NONE X
reciprocal square root estimate	RSQREST	16	32	64	128	NONE X
square root	SQR	16	32	64	128	NONE C F N X Z
sum	SUM	16	32	64	128	NONE C F N X Z

Format

E.op.prec.round rd=rc

rd=eopprecround(rc)

31 24	23 18	17 12	11 6	5 0
E.prec	rd	rc	op	E.UNARY
8	6	6	6	6

Description

The contents of register rc is used as the operand of the specified floating-point operation. The result is placed in register rd.

The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, unless default exception handling is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified or if default exception handling is specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

The reciprocal estimate and reciprocal square root estimate instructions compute an exact result for half precision, and a result with at least 12 bits of significant precision for larger formats.

Definition

```
def EnsembleUnaryFloatingPoint(op,prec,round,rd,rc) as
    c ← RegRead(rc, 128)
    case op of
        E.ABS.F, E.NEG.F, E.SQR.F:
            for i ← 0 to 128-prec by prec
                ci ← F(prec,$c_{i+prec-1..i}$)
                case op of
                    E.ABS.F:
                        ai.t ← ci.t
                        ai.s ← 0
                        ai.e ← ci.e
                        ai.f ← ci.f
                    E.COPY.F:
                        ai ← ci
                    E.NEG.F:
                        ai.t ← ci.t
                        ai.s ← -ci.s
                        ai.e ← ci.e
                        ai.f ← ci.f
                    E.RECEST.F:
                        ai ← frecest(ci)
                    E.RSQREST.F:
                        ai ← frsqrest(ci)
                    E.SQR.F:
                        ai ← fsqr(ci)
                endcase
                $a_{i+prec-1..i}$ ← PackF(prec, ai, round)
            endfor
        E.SUM.F:
            p[0].t ← NULL
            for i ← 0 to 128-prec by prec
                p[i+prec] ← fadd(p[i], F(prec,$c_{i+prec-1..i}$))
            endfor
            a ← PackF(prec, p[128], round)
        E.SINK.F:
            for i ← 0 to 128-prec by prec
                ci ← F(prec,$c_{i+prec-1..i}$)
                $a_{i+prec-1..i}$ ← fsinkr(prec, ci, round)
            endfor
        E.FLOAT.F:
            for i ← 0 to 128-prec by prec
                ci.t ← NORM
                ci.e ← 0
                ci.s ← $c_{i+prec-1}$
                ci.f ← ci.s ? 1+-$c_{i+prec-2..i}$ : $c_{i+prec-2..i}$
                $a_{i+prec-1..i}$ ← PackF(prec, ci, round)
            endfor
        E.INFLATE.F:
            for i ← 0 to 64-prec by prec
                ci ← F(prec,$c_{i+prec-1..i}$)
                $a_{i+i+prec+prec-1..i+i}$ ← PackF(prec+prec, ci, round)
            endfor
        E.DEFLATE.F:
            for i ← 0 to 128-prec by prec
                ci ← F(prec,$c_{i+prec-1..i}$)
                $a_{i/2+prec/2-1..i/2}$ ← PackF(prec/2, ci, round)
            endfor
            $a_{127..64}$ ← 0
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating point arithmetic

Wide Multiply Matrix

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

W.MUL.MAT.8.B	Wide multiply matrix signed byte big-endian
W.MUL.MAT.8.L	Wide multiply matrix signed byte little-endian
W.MUL.MAT.16.B	Wide multiply matrix signed doublet big-endian
W.MUL.MAT.16.L	Wide multiply matrix signed doublet little-endian
W.MUL.MAT.32.B	Wide multiply matrix signed quadlet big-endian
W.MUL.MAT.32.L	Wide multiply matrix signed quadlet little-endian
W.MUL.MAT.C.8.B	Wide multiply matrix signed complex byte big-endian
W.MUL.MAT.C.8.L	Wide multiply matrix signed complex byte little-endian
W.MUL.MAT.C.16.B	Wide multiply matrix signed complex doublet big-endian
W.MUL.MAT.C.16.L	Wide multiply matrix signed complex doublet little-endian
W.MUL.MAT.M.8.B	Wide multiply matrix mixed-signed byte big-endian
W.MUL.MAT.M.8.L	Wide multiply matrix mixed-signed byte little-endian
W.MUL.MAT.M.16.B	Wide multiply matrix mixed-signed doublet big-endian
W.MUL.MAT.M.16.L	Wide multiply matrix mixed-signed doublet little-endian
W.MUL.MAT.M.32.B	Wide multiply matrix mixed-signed quadlet big-endian
W.MUL.MAT.M.32.L	Wide multiply matrix mixed-signed quadlet little-endian
W.MUL.MAT.P.8.B	Wide multiply matrix polynomial byte big-endian
W.MUL.MAT.P.8.L	Wide multiply matrix polynomial byte little-endian
W.MUL.MAT.P.16.B	Wide multiply matrix polynomial doublet big-endian
W.MUL.MAT.P.16.L	Wide multiply matrix polynomial doublet little-endian
W.MUL.MAT.P.32.B	Wide multiply matrix polynomial quadlet big-endian
W.MUL.MAT.P.32.L	Wide multiply matrix polynomial quadlet little-endian
W.MUL.MAT.U.8.B	Wide multiply matrix unsigned byte big-endian
W.MUL.MAT.U.8.L	Wide multiply matrix unsigned byte little-endian
W.MUL.MAT.U.16.B	Wide multiply matrix unsigned doublet big-endian
W.MUL.MAT.U.16.L	Wide multiply matrix unsigned doublet little-endian
W.MUL.MAT.U.32.B	Wide multiply matrix unsigned quadlet big-endian
W.MUL.MAT.U.32.L	Wide multiply matrix unsigned quadlet little-endian

Selection

class	op	type	size	order
multiply	W.MUL.MAT.	NONE M U P	8 16 32	B L
		C	8 16	B L

Format

W.op.size.order rd=rc,rb

rd=wopsizeorder(rc,rb)

31 24	23 18	17 12	11 6	5 4	3 0
W.MINOR.order	rd	rc	rb	sz	op
8	6	6	6	2	4

Description

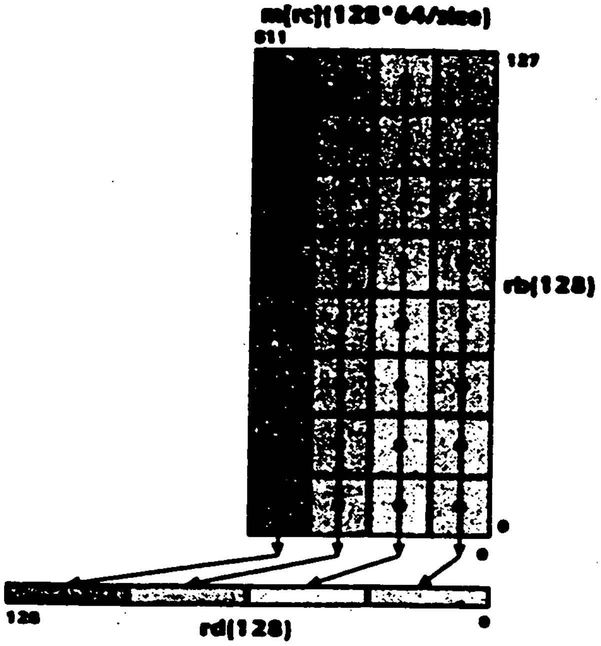
The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is catenated and placed in register rd.

The memory-multiply instructions (W.MUL.MAT, W.MUL.MAT.C, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) perform a partitioned array multiply of up to 8192 bits, that is 64x128 bits. The width of the array can be limited to 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

The virtual address must either be aligned to 1024/gsize bytes (or 512/gsize for W.MUL.MAT.C) (with gsize measured in bits), or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one-quarter of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

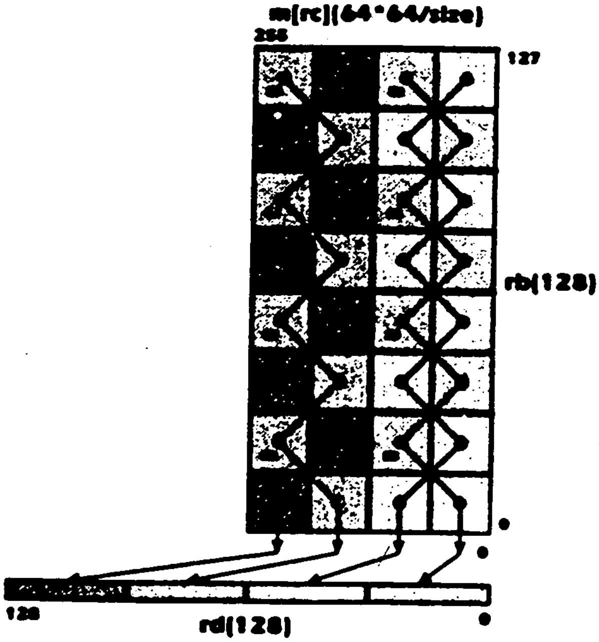
A wide-multiply-octlets instruction (W.MUL.MAT.type.64, type=NONE M U P) is not implemented and causes a reserved instruction exception, as an ensemble-multiply-sum-octlets instruction (E.MUL.SUM.type.64) performs the same operation except that the multiplier is sourced from a 128-bit register rather than memory. Similarly, instead of wide-multiply-complex-quadlets instruction (W.MUL.MAT.C.32), one should use an ensemble-multiply-complex-quadlets instruction (E.MUL.SUM.C.32).

A wide-multiply-doublets instruction (W.MUL.MAT, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) multiplies memory [m31 m30 ... m1 m0] with vector [h g f e d c b a], yielding products [hm31+gm27+...+bm7+am3 ... hm28+gm24+...+bm4+am0]:



Wide multiply matrix

A wide-multiply-matrix-complex-doublets instruction (W.MUL.MAT.C) multiplies memory [m15 m14 ... m1 m0] with vector [h g f e d c b a], yielding products [hm14+gm15+...+bm2+am3 ... hm12+gm13+...+bm0+am1 -hm13+gm12+...-bm1+am0]:



Wide multiply matrix complex

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← $((vs\&v_{size-1+i})^{h-size} \mid\mid v_{size-1+i..i}) * ((ws\&w_{size-1+j})^{h-size} \mid\mid w_{size-1+j..j})$
enddef

def c ← PolyMultiply(size,a,b) as
    p[0] ← $0^{2*size}$
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ $a_k$ ? $(0^{size-k} \mid\mid b \mid\mid 0^k)$ : $0^{2*size}$
    endfor
    c ← p[size]
enddef

def MemoryMultiply(major,op,gsize,rd,rc,rb)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    if $c_{lgsize-4..0} \neq 0$ then
        raise AccessDisallowedByVirtualAddress
    endif
    if $c_{2..lgsize-3} \neq 0$ then
        wsize ← $(c \text{ and } (0-c)) \mid\mid 0^4$
        t ← c and (c-1)
    else
        wsize ← 64
        t ← a
    endif
    lwsize ← log(wsize)
    if $t_{lwsize+6-lgsize..lwsize-3} \neq 0$ then
        msize ← $(t \text{ and } (0-t)) \mid\mid 0^4$
        VirtAddr ← t and (t-1)
    else
        msize ← 128*wsize/gsize
        VirtAddr ← t
    endif
    case major of
        W.MINOR.B:
            order ← B
        W.MINOR.L:
            order ← L
    endcase
    case op of
        W.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64:
            ms ← bs ← 0
        W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32, W.MUL.MAT.M.64:
            ms ← 0
            bs ← 1
        W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32, W.MUL.MAT.64,
        W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32, W.MUL.MAT.C.64:
            ms ← bs ← 1
        W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32, W.MUL.MAT.P.64:
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    h ← 2*gsize
    for i ← 0 to wsize-gsize by gsize
        q[0] ← $0^{2*gsize}$
        for j ← 0 to vsize-gsize by gsize
            case op of
                W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32, W.MUL.MAT.P.64:
                    k ← i+wsize*$j_{8..lgsize}$
                    q[j+gsize] ← q[j] ^ PolyMultiply(gsize,$m_{k+gsize-1..k}$,$b_{j+gsize-1..j}$)
                W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32, W.MUL.MAT.C.64:
                    if (~i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,k,bs,b,j)
                    else
                        k ← i+gsize+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← q[j] - mul(gsize,h,ms,m,k,bs,b,j)
                    endif
                W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32, W.MUL.MAT.64,
                W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32, W.MUL.MAT.M.64,
                W.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64:
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+wsize*$j_{8..lgsize}$,bs,b,j)
        endfor
        $a_{2*gsize-1+2*i..2*i}$ ← q[vsize]
    endfor
    $a_{127..2*wsize}$ ← 0
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Wide Multiply Matrix Extract

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register.

Operation codes

W.MUL.MAT.X.B	Wide multiply matrix extract big-endian
W.MUL.MAT.X.L	Wide multiply matrix extract little-endian

Format

op ra=rc,rd,rb

ra=op(rc,rd,rb)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

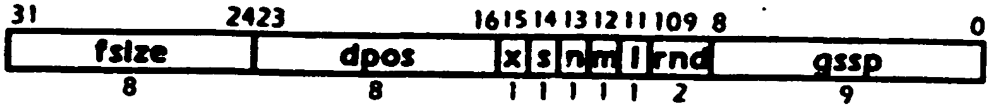
The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rd. The group size and other parameters are specified from the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed, producing a group of values. The group of values is rounded, and limited as specified, yielding a group of results which is the size specified. The group of results is catenated and placed in register ra.

NOTE: The size of this operation is determined from the contents of register rb The multiplier usage is constant, but the memory operand size is inversely related to the group size. Presumably this can be checked for cache validity.

We also use low order bits of rc to designate a size, which must be consistent with the group size. Because the memory operand is cached, the size can also be cached, thus eliminating the time required to decode the size, whether from rb or from rc.

The wide-multiply-matrix-extract instructions (W.MUL.MAT.X.B, W.MUL.MAT.X.L) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

Bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted. The position and default values of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI instruction.



The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	mixed-sign vs. same-sign multiplication
l	1	saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

The values in the **s**, **n**, **m**, **t**, and **rnd** fields have the following meaning:

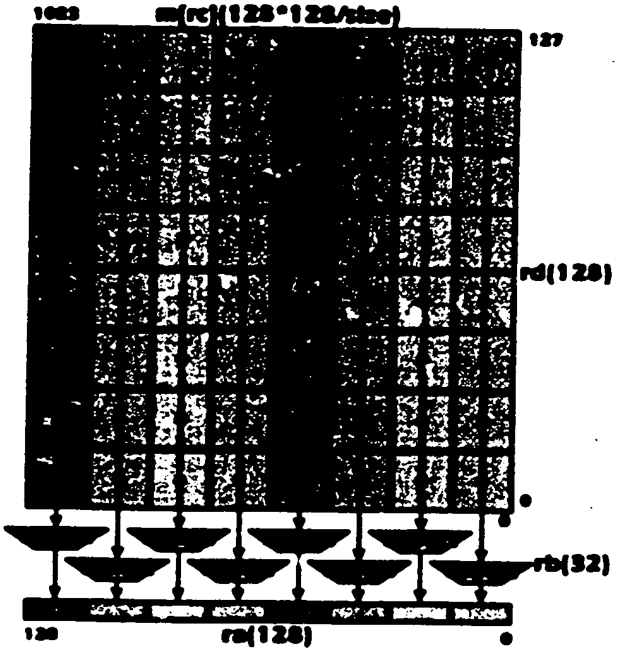
values	s	n	m	l	rnd
0	unsigned	real	same-sign	truncate	F
1	signed	complex	mixed-sign	saturate	Z
2					N
3					C

The virtual address must be aligned, that is, it must be an exact multiple of the operand size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

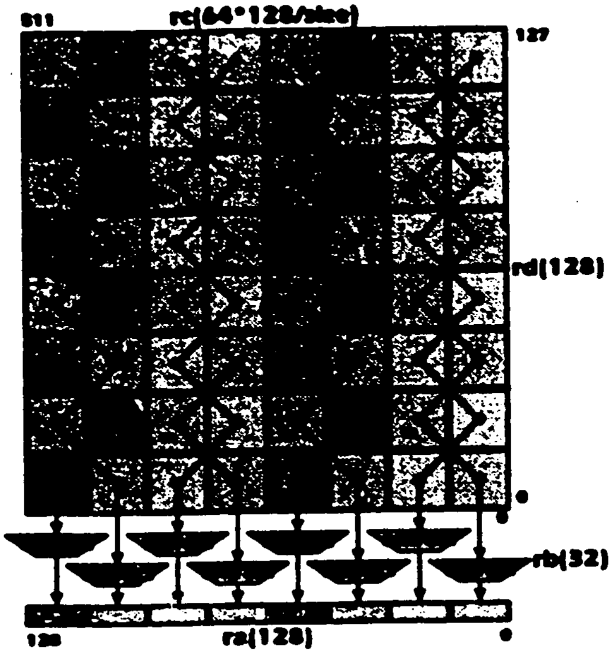
A wide-multiply-matrix-extract-doublets instruction (W.MUL.MAT.X.B or W.MUL.MAT.X.L) multiplies memory [m63 m62 m61 … m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm15+cm23+dm31+em39+fm47+gm55+hm63 … am2+bm10+cm18+dm26+em34+fm42+gm50+hm58

am1+bm9+cm17+dm25+em33+fm41+gm49+hm57
am0+bm8+cm16+dm24+em32+fm40+gm48+hm56], rounded and limited as specified:



Wide multiply extract matrix doublets

A wide-multiply-matrix-extract-complex-doublets instruction (W.MUL.MAT.X with n set in rb) multiplies memory [m31 m30 m29 … m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 … am2-bm3+cm10-dm11+em18-fm19+gm26-hm27 am1+bm0+cm9+dm8+em17+fm16+gm25+hm24 am0-bm1+cm8-dm9+em16-f17+gm24-hm25], rounded and limited as specified:



Wide multiply extract matrix complex doublets

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← $((vs\&v_{size-1+i})^{h-size} \,||\, v_{size-1+i..i}) * ((ws\&w_{size-1+j})^{h-size} \,||\, w_{size-1+j..j})$
enddef

def WideMultiplyExtractMatrix(op,ra,rb,rc,rd)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    case $b_{8..0}$ of
        0..255:
            sgsize ← 128
        256..383:
            sgsize ← 64
        384..447:
            sgsize ← 32
        448..479:
            sgsize ← 16
        480..495:
            sgsize ← 8
        496..503:
            sgsize ← 4
```

```
          504..507:
               sgsize ← 2
          508..511:
               sgsize ← 1
     endcase
     l ← $b_{11}$
     m ← $b_{12}$
     n ← $b_{13}$
     signed ← $b_{14}$
     if $c_{3..0}$ ≠ 0 then
          wsize ← (c and (0-c)) || $0^4$
          t ← c and (c-1)
     else
          wsize ← 128
          t ← c
     endif
     if sgsize < 8 then
          gsize ← 8
     elseif sgsize > wsize/2 then
          gsize ← wsize/2
     else
          gsize ← sgsize
     endif
     lgsize ← log(gsize)
     lwsize ← log(wsize)
     if $t_{lwsize+6-n-lgsize..lwsize-3}$ ≠ 0 then
          msize ← (t and (0-t)) || $0^4$
          VirtAddr ← t and (t-1)
     else
          msize ← 64*(2-n)*wsize/gsize
          VirtAddr ← t
     endif
     vsize ← (1+n)*msize*gsize/wsize
     mm ← LoadMemory(c,VirtAddr,msize,order)
     h ← (2*gsize) + 7 - lgsize
     lmsize ← log(msize)
     if (VirtAddr$_{lmsize-4..0}$ ≠ 0 then
          raise AccessDisallowedByVirtualAddress
     endif
     case op of
          W.MUL.MAT.X.B:
               order ← B
          W.MUL.MAT.X.L:
               order ← L
     endcase
     ms ← signed
     ds ← signed ^ m
     as ← signed or m
     spos ← ($b_{8..0}$) and (2*gsize-1)
     dpos ← (0 || $b_{23..16}$) and (gsize-1)
     r ← spos
     sfsize ← (0 || $b_{31..24}$) and (gsize-1)
```

```
    tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
    fsize ← (tfsize + spos > h) ? h - spos : tfsize
    if ($b_{10..9}$ = Z) & ~signed then
        rnd ← F
    else
        rnd ← $b_{10..9}$
    endif
    for i ← 0 to wsize-gsize by gsize
        q[0] ← $0^{2*gsize+7-lgsize}$
        for j ← 0 to vsize-gsize by gsize
            if n then
                if (~i) & j & gsize = 0 then
                    k ← i-(j&gsize)+wsize*$j_{8..lgsize+1}$
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,mm,k,ds,d,j)
                else
                    k ← i+gsize+wsize*$j_{8..lgsize+1}$
                    q[j+gsize] ← q[j] - mul(gsize,h,ms,mm,k,ds,d,j)
                endif
            else
                q[j+gsize] ← q[j] + mul(gsize,h,ms,mm,i+j*wsize/gsize,ds,d,j)
            endif
        endfor
        p ← q[128]
        case rnd of
            none, N:
                s ← $0^{h-r}$ || ~$p_r$ || $p_r^{r-1}$
            Z:
                s ← $0^{h-r}$ || $p_{h-1}^r$
            F:
                s ← $0^h$
            C:
                s ← $0^{h-r}$ || $1^r$
        endcase
        v ← ((ds & $p_{h-1}$)||p) + (0||s)
        if ($v_{h..r+fsize}$ = (as & $v_{r+fsize-1}$)$^{h+1-r-fsize}$) or not l then
            w ← (as & $v_{r+fsize-1}$)$^{gsize-fsize-dpos}$ || $v_{fsize-1+r..r}$ || $0^{dpos}$
        else
            w ← (s ? ($v_h$ || ~$v_h^{gsize-dpos-1}$) : $1^{gsize-dpos}$) || $0^{dpos}$
        endif
        $a_{size-1+i..i}$ ← w
    endfor
    $a_{127..wsize}$ ← 0
    RegWrite(ra, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB

Access detail required by global TB
Local TB miss
Global TB miss

Wide Multiply Matrix Extract Immediate

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

W.MUL.MAT.X.I.8.C.B	Wide multiply matrix extract immediate signed bytes big endian ceiling
W.MUL.MAT.X.I.8.C.L	Wide multiply matrix extract immediate signed bytes little endian ceiling
W.MUL.MAT.X.I.8.F.B	Wide multiply matrix extract immediate signed bytes big endian floor
W.MUL.MAT.X.I.8.F.L	Wide multiply matrix extract immediate signed bytes little endian floor
W.MUL.MAT.X.I.8.N.B	Wide multiply matrix extract immediate signed bytes big endian nearest
W.MUL.MAT.X.I.8.N.L	Wide multiply matrix extract immediate signed bytes little endian nearest
W.MUL.MAT.X.I.8.Z.B	Wide multiply matrix extract immediate signed bytes big endian zero
W.MUL.MAT.X.I.8.Z.L	Wide multiply matrix extract immediate signed bytes little endian zero
W.MUL.MAT.X.I.16.C.B	Wide multiply matrix extract immediate signed doublets big endian ceiling
W.MUL.MAT.X.I.16.C.L	Wide multiply matrix extract immediate signed doublets little endian ceiling
W.MUL.MAT.X.I.16.F.B	Wide multiply matrix extract immediate signed doublets big endian floor
W.MUL.MAT.X.I.16.F.L	Wide multiply matrix extract immediate signed doublets little endian floor
W.MUL.MAT.X.I.16.N.B	Wide multiply matrix extract immediate signed doublets big endian nearest
W.MUL.MAT.X.I.16.N.L	Wide multiply matrix extract immediate signed doublets little endian nearest
W.MUL.MAT.X.I.16.Z.B	Wide multiply matrix extract immediate signed doublets big endian zero
W.MUL.MAT.X.I.16.Z.L	Wide multiply matrix extract immediate signed doublets little endian zero
W.MUL.MAT.X.I.32.C.B	Wide multiply matrix extract immediate signed quadlets big endian ceiling
W.MUL.MAT.X.I.32.C.L	Wide multiply matrix extract immediate signed quadlets little endian ceiling
W.MUL.MAT.X.I.32.F.B	Wide multiply matrix extract immediate signed quadlets big endian floor
W.MUL.MAT.X.I.32.F.L	Wide multiply matrix extract immediate signed quadlets little endian floor
W.MUL.MAT.X.I.32.N.B	Wide multiply matrix extract immediate signed quadlets big endian nearest
W.MUL.MAT.X.I.32.N.L	Wide multiply matrix extract immediate signed quadlets little endian nearest
W.MUL.MAT.X.I.32.Z.B	Wide multiply matrix extract immediate signed quadlets big endian zero
W.MUL.MAT.X.I.32.Z.L	Wide multiply matrix extract immediate signed quadlets little endian zero
W.MUL.MAT.X.I.64.C.B	Wide multiply matrix extract immediate signed octlets big endian ceiling
W.MUL.MAT.X.I.64.C.L	Wide multiply matrix extract immediate signed octlets little endian ceiling
W.MUL.MAT.X.I.64.F.B	Wide multiply matrix extract immediate signed octlets big endian floor
W.MUL.MAT.X.I.64.F.L	Wide multiply matrix extract immediate signed octlets little endian floor
W.MUL.MAT.X.I.64.N.B	Wide multiply matrix extract immediate signed octlets big endian nearest
W.MUL.MAT.X.I.64.N.L	Wide multiply matrix extract immediate signed octlets little endian nearest
W.MUL.MAT.X.I.64.Z.B	Wide multiply matrix extract immediate signed octlets big endian zero
W.MUL.MAT.X.I.64.Z.L	Wide multiply matrix extract immediate signed octlets little endian zero
W.MUL.MAT.X.I.C.8.C.B	Wide multiply matrix extract immediate complex bytes big endian ceiling
W.MUL.MAT.X.I.C.8.C.L	Wide multiply matrix extract immediate complex bytes little endian ceiling
W.MUL.MAT.X.I.C.8.F.B	Wide multiply matrix extract immediate complex bytes big endian floor
W.MUL.MAT.X.I.C.8.F.L	Wide multiply matrix extract immediate complex bytes little endian floor
W.MUL.MAT.X.I.C.8.N.B	Wide multiply matrix extract immediate complex bytes big endian nearest
W.MUL.MAT.X.I.C.8.N.L	Wide multiply matrix extract immediate complex bytes little endian nearest
W.MUL.MAT.X.I.C.8.Z.B	Wide multiply matrix extract immediate complex bytes big endian zero
W.MUL.MAT.X.I.C.8.Z.L	Wide multiply matrix extract immediate complex bytes little endian zero

W.MUL.MAT.X.I.C.16.C.B	Wide multiply matrix extract immediate complex doublets big-endian ceiling
W.MUL.MAT.X.I.C.16.C.L	Wide multiply matrix extract immediate complex doublets little-endian ceiling
W.MUL.MAT.X.I.C.16.F.B	Wide multiply matrix extract immediate complex doublets big-endian floor
W.MUL.MAT.X.I.C.16.F.L	Wide multiply matrix extract immediate complex doublets little-endian floor
W.MUL.MAT.X.I.C.16.N.B	Wide multiply matrix extract immediate complex doublets big-endian nearest
W.MUL.MAT.X.I.C.16.N.L	Wide multiply matrix extract immediate complex doublets little-endian nearest
W.MUL.MAT.X.I.C.16.Z.B	Wide multiply matrix extract immediate complex doublets big-endian zero
W.MUL.MAT.X.I.C.16.Z.L	Wide multiply matrix extract immediate complex doublets little-endian zero
W.MUL.MAT.X.I.C.32.C.B	Wide multiply matrix extract immediate complex quadlets big-endian ceiling
W.MUL.MAT.X.I.C.32.C.L	Wide multiply matrix extract immediate complex quadlets little-endian ceiling
W.MUL.MAT.X.I.C.32.F.B	Wide multiply matrix extract immediate complex quadlets big-endian floor
W.MUL.MAT.X.I.C.32.F.L	Wide multiply matrix extract immediate complex quadlets little-endian floor
W.MUL.MAT.X.I.C.32.N.B	Wide multiply matrix extract immediate complex quadlets big-endian nearest
W.MUL.MAT.X.I.C.32.N.L	Wide multiply matrix extract immediate complex quadlets little-endian nearest
W.MUL.MAT.X.I.C.32.Z.B	Wide multiply matrix extract immediate complex quadlets big-endian zero
W.MUL.MAT.X.I.C.32.Z.L	Wide multiply matrix extract immediate complex quadlets little-endian zero
W.MUL.MAT.X.I.C.64.C.B	Wide multiply matrix extract immediate complex octlets big-endian ceiling
W.MUL.MAT.X.I.C.64.C.L	Wide multiply matrix extract immediate complex octlets little-endian ceiling
W.MUL.MAT.X.I.C.64.F.B	Wide multiply matrix extract immediate complex octlets big-endian floor
W.MUL.MAT.X.I.C.64.F.L	Wide multiply matrix extract immediate complex octlets little-endian floor
W.MUL.MAT.X.I.C.64.N.B	Wide multiply matrix extract immediate complex octlets big-endian nearest
W.MUL.MAT.X.I.C.64.N.L	Wide multiply matrix extract immediate complex octlets little-endian nearest
W.MUL.MAT.X.I.C.64.Z.B	Wide multiply matrix extract immediate complex octlets big-endian zero
W.MUL.MAT.X.I.C.64.Z.L	Wide multiply matrix extract immediate complex octlets little-endian zero
W.MUL.MAT.X.I.M.8.C.B	Wide multiply matrix extract immediate mixed-signed bytes big-endian ceiling
W.MUL.MAT.X.I.M.8.C.L	Wide multiply matrix extract immediate mixed-signed bytes little-endian ceiling
W.MUL.MAT.X.I.M.8.F.B	Wide multiply matrix extract immediate mixed-signed bytes big-endian floor
W.MUL.MAT.X.I.M.8.F.L	Wide multiply matrix extract immediate mixed-signed bytes little-endian floor
W.MUL.MAT.X.I.M.8.N.B	Wide multiply matrix extract immediate mixed-signed bytes big-endian nearest
W.MUL.MAT.X.I.M.8.N.L	Wide multiply matrix extract immediate mixed-signed bytes little-endian nearest
W.MUL.MAT.X.I.M.8.Z.B	Wide multiply matrix extract immediate mixed-signed bytes big-endian zero
W.MUL.MAT.X.I.M.8.Z.L	Wide multiply matrix extract immediate mixed-signed bytes little-endian zero
W.MUL.MAT.X.I.M.16.C.B	Wide multiply matrix extract immediate mixed-signed doublets big-endian ceiling
W.MUL.MAT.X.I.M.16.C.L	Wide multiply matrix extract immediate mixed-signed doublets little-endian ceiling
W.MUL.MAT.X.I.M.16.F.B	Wide multiply matrix extract immediate mixed-signed doublets big-endian floor
W.MUL.MAT.X.I.M.16.F.L	Wide multiply matrix extract immediate mixed-signed doublets little-endian floor
W.MUL.MAT.X.I.M.16.N.B	Wide multiply matrix extract immediate mixed-signed doublets big-endian nearest
W.MUL.MAT.X.I.M.16.N.L	Wide multiply matrix extract immediate mixed-signed doublets little-endian nearest
W.MUL.MAT.X.I.M.16.Z.B	Wide multiply matrix extract immediate mixed-signed doublets big-endian zero
W.MUL.MAT.X.I.M.16.Z.L	Wide multiply matrix extract immediate mixed-signed doublets little-endian zero
W.MUL.MAT.X.I.M.32.C.B	Wide multiply matrix extract immediate mixed-signed quadlets big-endian ceiling
W.MUL.MAT.X.I.M.32.C.L	Wide multiply matrix extract immediate mixed-signed quadlets little-endian ceiling
W.MUL.MAT.X.I.M.32.F.B	Wide multiply matrix extract immediate mixed-signed quadlets big-endian floor
W.MUL.MAT.X.I.M.32.F.L	Wide multiply matrix extract immediate mixed-signed quadlets little-endian floor
W.MUL.MAT.X.I.M.32.N.B	Wide multiply matrix extract immediate mixed-signed quadlets big-endian nearest
W.MUL.MAT.X.I.M.32.N.L	Wide multiply matrix extract immediate mixed-signed quadlets little-endian nearest
W.MUL.MAT.X.I.M.32.Z.B	Wide multiply matrix extract immediate mixed-signed quadlets big-endian zero
W.MUL.MAT.X.I.M.32.Z.L	Wide multiply matrix extract immediate mixed-signed quadlets little-endian zero
W.MUL.MAT.X.I.M.64.C.B	Wide multiply matrix extract immediate mixed-signed octlets big-endian ceiling
W.MUL.MAT.X.I.M.64.C.L	Wide multiply matrix extract immediate mixed-signed octlets little-endian ceiling

W.MUL.MAT.X.I.M.64.F.B	Wide multiply matrix extract immediate mixed-signed octlets big-endian floor
W.MUL.MAT.X.I.M.64.F.L	Wide multiply matrix extract immediate mixed-signed octlets little-endian floor
W.MUL.MAT.X.I.M.64.N.B	Wide multiply matrix extract immediate mixed-signed octlets big-endian nearest
W.MUL.MAT.X.I.M.64.N.L	Wide multiply matrix extract immediate mixed-signed octlets little-endian nearest
W.MUL.MAT.X.I.M.64.Z.B	Wide multiply matrix extract immediate mixed-signed octlets big-endian zero
W.MUL.MAT.X.I.M.64.Z.L	Wide multiply matrix extract immediate mixed-signed octlets little-endian zero
W.MUL.MAT.X.I.U.8.C.B	Wide multiply matrix extract immediate unsigned bytes big-endian ceiling
W.MUL.MAT.X.I.U.8.C.L	Wide multiply matrix extract immediate unsigned bytes little-endian ceiling
W.MUL.MAT.X.I.U.8.F.B	Wide multiply matrix extract immediate unsigned bytes big-endian floor
W.MUL.MAT.X.I.U.8.F.L	Wide multiply matrix extract immediate unsigned bytes little-endian floor
W.MUL.MAT.X.I.U.8.N.B	Wide multiply matrix extract immediate unsigned bytes big-endian nearest
W.MUL.MAT.X.I.U.8.N.L	Wide multiply matrix extract immediate unsigned bytes little-endian nearest
W.MUL.MAT.X.I.U.16.C.B	Wide multiply matrix extract immediate unsigned doublets big-endian ceiling
W.MUL.MAT.X.I.U.16.C.L	Wide multiply matrix extract immediate unsigned doublets little-endian ceiling
W.MUL.MAT.X.I.U.16.F.B	Wide multiply matrix extract immediate unsigned doublets big-endian floor
W.MUL.MAT.X.I.U.16.F.L	Wide multiply matrix extract immediate unsigned doublets little-endian floor
W.MUL.MAT.X.I.U.16.N.B	Wide multiply matrix extract immediate unsigned doublets big-endian nearest
W.MUL.MAT.X.I.U.16.N.L	Wide multiply matrix extract immediate unsigned doublets little-endian nearest
W.MUL.MAT.X.I.U.32.C.B	Wide multiply matrix extract immediate unsigned quadlets big-endian ceiling
W.MUL.MAT.X.I.U.32.C.L	Wide multiply matrix extract immediate unsigned quadlets little-endian ceiling
W.MUL.MAT.X.I.U.32.F.B	Wide multiply matrix extract immediate unsigned quadlets big-endian floor
W.MUL.MAT.X.I.U.32.F.L	Wide multiply matrix extract immediate unsigned quadlets little-endian floor
W.MUL.MAT.X.I.U.32.N.B	Wide multiply matrix extract immediate unsigned quadlets big-endian nearest
W.MUL.MAT.X.I.U.32.N.L	Wide multiply matrix extract immediate unsigned quadlets little-endian nearest
W.MUL.MAT.X.I.U.64.C.B	Wide multiply matrix extract immediate unsigned octlets big-endian ceiling
W.MUL.MAT.X.I.U.64.C.L	Wide multiply matrix extract immediate unsigned octlets little-endian ceiling
W.MUL.MAT.X.I.U.64.F.B	Wide multiply matrix extract immediate unsigned octlets big-endian floor
W.MUL.MAT.X.I.U.64.F.L	Wide multiply matrix extract immediate unsigned octlets little-endian floor
W.MUL.MAT.X.I.U.64.N.B	Wide multiply matrix extract immediate unsigned octlets big-endian nearest
W.MUL.MAT.X.I.U.64.N.L	Wide multiply matrix extract immediate unsigned octlets little-endian nearest

Format

W.op.size.rnd rd=rc,rb,i

rd=wopsizernd(rc,rb,i)

31 24	23 18	17 12	11 6	5 4	3 2	1 0
op	rd	rc	rb	sz	rnd	sh
8	6	6	6	2	2	2

```
sz ← log(size) - 3
case op of
    W.MULMATX.I, W.MULMATX.I.C:
        assert size + 6 - log(size) ≥ i ≥ size + 6 - log(size) - 3
        sh ← size + 6 - log(size) - i
    W.MULMATX.I.M, W.MULMATX.I.U:
        assert size + 7 - log(size) ≥ i ≥ size + 7 - log(size) - 3
        sh ← size + 7 - log(size) - i
endcase
```

Description

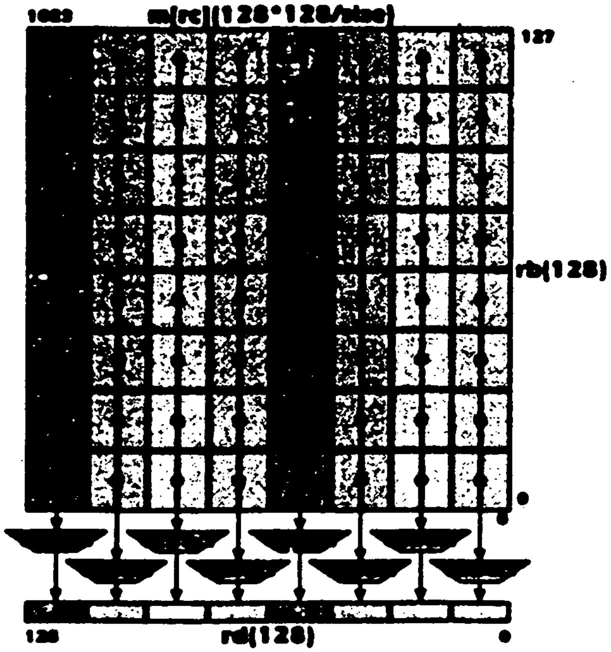
The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed, or are convolved, producing a group of sums. The group of sums is rounded, and limited as specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

The wide-multiply-extract-immediate-matrix instructions (W.MUL.MAT.X.I, W.MUL.MAT.X.I.U, W.MUL.MAT.X.I.M, W.MUL.MAT.X.I.C) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

The virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.X.I.C), or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

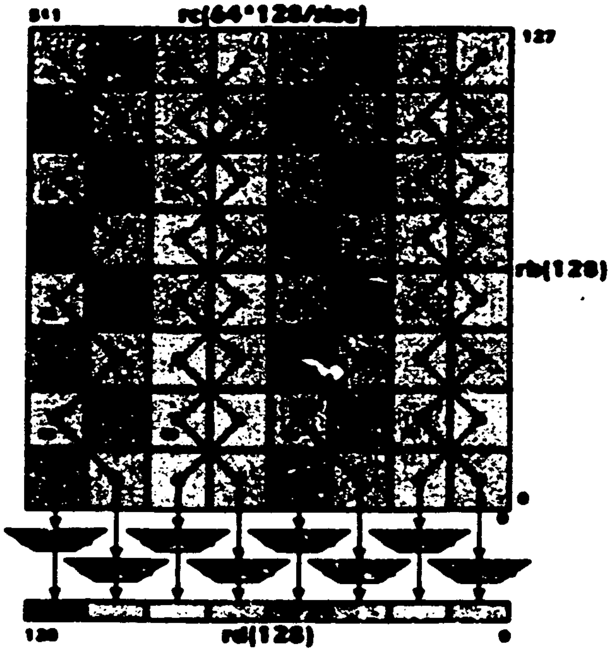
A wide-multiply-extract-immediate-matrix-doublets instruction (W.MUL.MAT.X.I.16 or W.MUL.MAT.X.I.U.16) multiplies memory [m63 m62 m61 ... m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm15+cm23+dm31+em39+fm47+gm55+hm63 ...
am2+bm10+cm18+dm26+em34+fm42+gm50+hm58
am1+bm9+cm17+dm25+em33+fm41+gm49+hm57
am0+bm8+cm16+dm24+em32+fm40+gm48+hm56], rounded and limited as specified:



Wide multiply matrix extract immediate doublets

/

A wide-multiply-matrix-extract-immediate-complex-doublets instruction (W.MUL.MAT.X.I.C.16) multiplies memory [m31 m30 m29 … m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 … am2-bm3+cm10-dm11+em18-fm19+gm26-hm27
am1+bm0+cm9+dm8+em17+fm16+gm25+hm24 am0-bm1+cm8-dm9+em16-f17+gm24-hm25], rounded and limited as specified:



Wide multiply matrix extract immediate complex doublets

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← $((vs\&v_{size-1+i})^{h-size} \mid\mid v_{size-1+i..i}) * ((vs\&w_{size-1+j})^{h-size} \mid\mid w_{size-1+j..j})$
enddef

def WideMultiplyExtractImmediateMatrix(op,md,gsize,rd,rc,rb,sh)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    case op of
        W.MUL.MAT.X.I.B, W.MUL.MAT.X.I.L, W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.U.L,
        W.MUL.MAT.X.I.M.B, W.MUL.MAT.X.I.M.L :
            if $c_{lgsize-4..0}$ ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if $c_{3..lgsize-3}$ ≠ 0 then
                wsize ← (c and (0-c)) || $0^4$
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if $t_{lwsize+6-lgsize..lwsize-3} \ne 0$ then
                msize ← (t and (0-t)) || $0^4$
                VirtAddr ← t and (t-1)
            else
                msize ← 128*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← msize*gsize/wsize
        W.MUL.MAT.X.I.C.B, W.MUL.MAT.X.I.C.L:
            if $c_{lgsize-4..0} \ne 0$ then
                raise AccessDisallowedByVirtualAddress
            endif
            if $c_{3..lgsize-3} \ne 0$ then
                wsize ← (c and (0-c)) || $0^4$
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if $t_{lwsize+5-lgsize..lwsize-3} \ne 0$ then
                msize ← (t and (0-t)) || $0^4$
                VirtAddr ← t and (t-1)
            else
                msize ← 64*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← 2*msize*gsize/wsize
    endcase
    case op of
        W.MUL.MAT.X.I.B, W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.M.B, W.MUL.MAT.X.I.C.B:
            order ← B
        W.MUL.MAT.X.I.L, W.MUL.MAT.X.I.U.L, W.MUL.MAT.X.I.M.L, W.MUL.MAT.X.I.C.L:
            order ← L
    endcase
    case op of
        W.MUL.MAT.X.I.U.B, W.MUL.MAT.X.I.U.L:
            as ← ms ← bs ← 0
        W.MUL.MAT.X.I.M.B, W.MUL.MAT.X.I.M.L:
            bs ← 0
            as ← ns ← 1
        W.MUL.MAT.X.I.B, W.MUL.MAT.X.I.L, W.MUL.MAT.X.I.C.B, W.MUL.MAT.X.I.C.L:
            as ← ms ← bs ← 1
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    h ← (2*gsize) + 7 - lgsize - (ms and bs)
    r ← h - size - sh
    for i ← 0 to wsize-gsize by gsize
        q[0] ← $0^{2*gsize+7-lgsize}$
        for j ← 0 to vsize-gsize by gsize
            case op of
```

```
                W.MULMAT.X.I.B, W.MULMAT.X.I.L
                W.MULMAT.X.I.U.B, W.MULMAT.X.I.U.L
                W.MULMAT.X.I.M.B, W.MULMAT.X.I.M.L:
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+wsize*$j_{8..lgsize}$,bs,b,j)
                W.MULMAT.X.I.C.B, W.MULMAT.X.I.C.L:
                    if (-i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,k,bs,b,j)
                    else
                        k ← i+gsize+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← q[j] - mul(gsize,h,ms,m,k,bs,b,j)
                    endif
            endcase
        endfor
        p ← q[vsize]
        case rnd of
            none, N:
                s ← $0^{h-r}$ || $-p_r$ || $p_r^{r-1}$
            Z:
                s ← $0^{h-r}$ || $p_{h-1}^r$
            F:
                s ← $0^h$
            C:
                s ← $0^{h-r}$ || $1^r$
        endcase
        v ← ((as & $p_{h-1}$)||p) + (0||s)
        if ($v_{h..r+gsize}$ = (as & $v_{r+gsize-1})^{h+1-r-gsize}$ then
            $a_{gsize-1+i..i}$ ← $v_{gsize-1+r..r}$
        else
            $a_{gsize-1+i..i}$ ← as ? ($v_h$ || $-v_h^{gsize-1}$) : $1^{gsize}$
        endif
    endfor
    $a_{127..wsize}$ ← 0
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Wide Multiply Matrix Floating-point

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

W.MULMAT.C.F.16.B	Wide multiply matrix complex floating-point half big-endian
W.MULMAT.C.F.16.L	Wide multiply matrix complex floating-point half little-endian
W.MULMAT.C.F.32.B	Wide multiply matrix complex floating-point single big-endian
W.MULMAT.C.F.32.L	Wide multiply matrix complex floating-point single little-endian
W.MULMAT.C.F.64.B	Wide multiply matrix complex floating-point double big-endian
W.MULMAT.C.F.64.L	Wide multiply matrix complex floating-point double little-endian
W.MULMAT.F.16.B	Wide multiply matrix floating-point half big-endian
W.MULMAT.F.16.L	Wide multiply matrix floating-point half little-endian
W.MULMAT.: .32.B	Wide multiply matrix floating-point single big-endian
W.MULMAT.F.32.L	Wide multiply matrix floating-point single little-endian
W.MULMAT.F.64.B	Wide multiply matrix floating-point double big-endian
W.MULMAT.F.64.L	Wide multiply matrix floating-point double little-endian

Format

M.op.size.order rd=rc,rb

rd=mopsizeorder(rc,rb)

31 24	23 18	17 12	11 6	5 4	3 0
W.MINOR.order	rd	rc	rb	sz	op
8	6	6	6	2	4

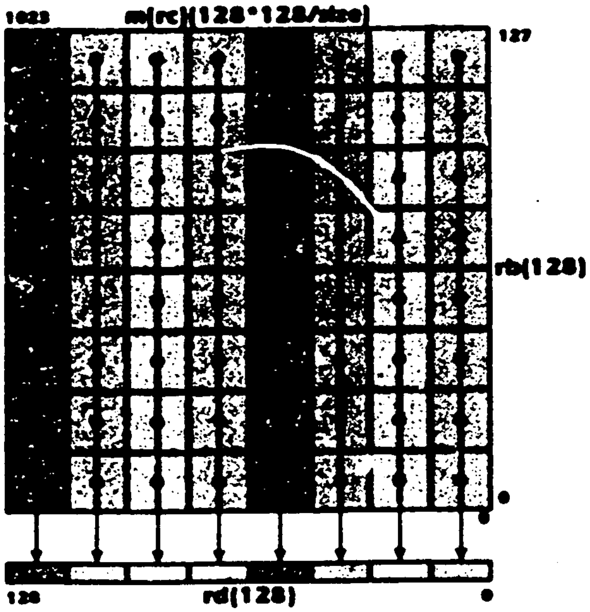
Description

The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is catenated and placed in register rd.

The wide-multiply-matrix-floating-point instructions (W.MUL.MAT.F, W.MUL.MAT.C.F) perform a partitioned array multuply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, or 2. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

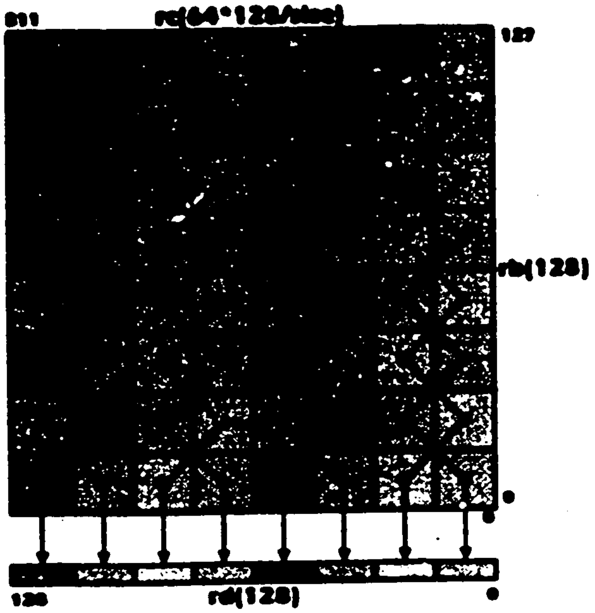
The virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.C.F), or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

A wide-multiply-matrix-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m31 m30 ... m1 m0] with vector [h g f e d c b a], yielding products [hm31+gm27+...+bm7+am3 ... hm28+gm24+...+bm4+am0]:



Wide multiply matrix floating-point half

A wide-multiply-matrix-complex-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m15 m14 ... m1 m0] with vector [h g f e d c b a], yielding products [hm14+gm15+...+bm2+am3 ... hm12+gm13+...+bm0+am1 -hm13+gm12+...-bm1+am0]:



Wide multiply matrix complex floating-point half

Definition

```
def mul(size,v,i,w,j) as
     mul ← fmul(F(size,v$_{size-1+i..i}$),F(size,w$_{size-1+j..j}$))
enddef

def MemoryFloatingPointMultiply(major,op,gsize,rd,rc,rb)
     c ← RegRead(rc, 64)
     b ← RegRead(rb, 128)
     lgsize ← log(gsize)
     switch op of
          W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
               if $c_{lgsize-4..0} \ne 0$ then
                    raise AccessDisallowedByVirtualAddress
               endif
               if $c_{3..lgsize-3} \ne 0$ then
                    wsize ← (c and (0-c)) || $0^4$
                    t ← c and (c-1)
               else
                    wsize ← 128
                    t ← c
               endif
               lwsize ← log(wsize)
               if $t_{lwsize+6-lgsize..lwsize-3} \ne 0$ then
                    msize ← (t and (0-t)) || $0^4$
                    VirtAddr ← t and (t-1)
               else
                    msize ← 128*wsize/gsize
                    VirtAddr ← t
               endif
```

```
            vsize ← msize*gsize/wsize
        W.MULMAT.C.F.16, W.MULMAT.C.F.32, W.MULMAT.C.F.64:
            if $c_{lgsize-4..0}$ ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if $c_{3..lgsize-3}$ ≠ 0 then
                wsize ← (c and (0-c)) || $0^4$
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if $t_{lwsize+5-lgsize..lwsize-3}$ ≠ 0 then
                msize ← (t and (0-t)) || $0^4$
                VirtAddr ← t and (t-1)
            else
                msize ← 64*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← 2*msize*gsize/wsize
    endcase
    case major of
        M.MINOR.B:
            order ← B
        M.MINOR.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    for i ← 0 to wsize-gsize by gsize
        q[0].t ← NULL
        for j ← 0 to vsize-gsize by gsize
            case op of
                W.MULMAT.F.16, W.MULMAT.F.32, W.MULMAT.F.64:
                    q[j+gsize] ← fadd(q[j], mul(gsize,m,i+wsize*$j_{8..lgsize}$,b,j))
                W.MULMAT.C.F.16, W.MULMAT.C.F.32, M.MULMAT.C.F.64:
                    if (~i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← fadd(q[j], mul(gsize,m,k,b,j))
                    else
                        k ← i+gsize+wsize*$j_{8..lgsize+1}$
                        q[j+gsize] ← fsub(q[j], mul(gsize,m,k,b,j))
                    endif
            endcase
        endfor
        $a_{gsize-1+i..i}$ ← q[vsize]
    endfor
    $a_{127..wsize}$ ← 0
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating-point arithmetic
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Wide Multiply Matrix Galois

These instructions take an address from a general register to fetch a large operand from memory, second and third operands from general registers, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register.

Operation codes

W.MUL.MAT.G.B	Wide multiply matrix Galois big-endian
W.MUL.MAT.G.L	Wide multiply matrix Galois little-endian

Format

W.MUL.MAT.G.order ra=rc,rd,rb

ra=mgmorder(rc,rd,rb)

31 24	23 18	17 12	11 6	5 0
W.MULG.order	rd	rc	rb	ra
8	6	6	6	6

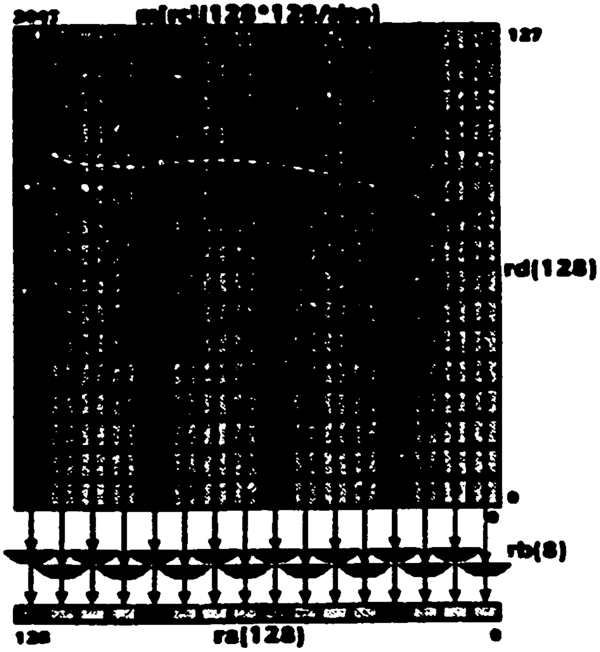
Description

The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. Second and third values are the contents of registers rd and rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied as polynomials with the first value, producing a result which is reduced to the Galois field specified by the third value, producing a group of result values. The group of result values is catenated and placed in register ra.

The wide-multiply-matrix-Galois instruction (W.MUL.MAT.G) performs a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired memory operand size in bytes to the virtual address operand.

The virtual address must either be aligned to 256 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

A wide-multiply-matrix-Galois instruction (W.MUL.MAT.G) multiplies memory [m255 m254 … m1 m0] with vector [p o n m l k j i h g f e d c b a], reducing the result modulo polynomial [q], yielding products [(pm255+om247+…+bm31+am15 mod q) (pm254+om246+…+bm30+am14 mod q) … (pm248+om240+…+bm16+am0 mod q)]:



Wide multiply matrix Galois

Definition

```
def c ← PolyMultiply(size,a,b) as
    p[0] ← $0^{2*size}$
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ $a_k$ ? ($0^{size-k}$ || b || $0^k$) : $0^{2*size}$
    endfor
    c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
    p[0] ← a
    for k ← size-1 to 0 by -1
        p[k+1] ← p[k] ^ $p[0]_{size+k}$ ? ($0^{size-k}$ || $1^1$ || b || $0^k$) : $0^{2*size}$
    endfor
    c ← $p[size]_{size-1..0}$
enddef

def WideMultiplyGalois(op,rd,rc,rb,ra)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    gsize ← 8
    lgsize ← log(gsize)
    if $c_{lgsize-4..0}$ ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if $c_{3..lgsize-3}$ ≠ 0 then
        wsize ← (c and (0-c)) || $0^4$
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    lwsize ← log(wsize)
    if $t_{lwsize+6-lgsize..lwsize-3}$ ≠ 0 then
        msize ← (t and (0-t)) || $0^4$
        VirtAddr ← t and (t-1)
    else
        msize ← 128*wsize/gsize
        VirtAddr ← t
    endif
    case op of
        W.MUL.MAT.G.B:
            order ← B
        W.MUL.MAT.G.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    for i ← 0 to wsize-gsize by gsize
        q[0] ← $0^{2*gsize}$
        for j ← 0 to vsize-gsize by gsize
            k ← i+wsize*$j_{8..lgsize}$
            q[j+gsize] ← q[j] ^ PolyMultiply(gsize,$m_{k+gsize-1..k}$,$d_{j+gsize-1..j}$)
        endfor
        $a_{gsize-1+i..i}$ ← PolyResidue(gsize,q[vsize],$b_{gsize-1..0}$)
    endfor
    $a_{127..wsize}$ ← 0
    RegWrite(ra, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Wide Switch

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

W.SWITCH.B	Wide switch big-endian
W.SWITCH.L	Wide switch little-endian

Format

op ra=rc,rd,rb

ra=op(rc,rd,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of register rc is specifies as a virtual address and optionally an operand size, and a value of specified size is loaded from memory. A second value is the catenated contents of registers rd and rb. Eight corresponding bits from the memory value are used to select a single result bit from the second value, for each corresponding bit position. The group of results is catenated and placed in register ra.

The virtual address must either be aligned to 128 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes. An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be 8, 16, 32, 64, or 128 bytes. If the address is not valid an "access disallowed by virtual address" exception occurs. When a size smaller than 128 bits is specified, the high order bits of the memory operand are replaced with values corresponding to the bit position, so that the same memory operand specifies a bit selection within symbols of the operand size, and the same operation is performed on each symbol.

Definition

```
def WideSwitch(op,rd,rc,rb,ra)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    if $c_{1..0} \ne 0$ then
        raise AccessDisallowedByVirtualAddress
    elseif $c_{6..0} \ne 0$ then
        VirtAddr ← c and (c-1)
        w ← wsize ← (c and (0-c)) || $0^1$
    else
        VirtAddr ← c
        w ← wsize ← 128
    endif
    msize ← 8*wsize
    lwsize ← log(wsize)
    case op of
        W.SWITCH.B:
            order ← B
        W.SWITCH.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    db ← d || b
    for i ← 0 to 127
        j ← 0 || $i_{lwsize-1..0}$
        k ← $m_{7*w+j}$ || $m_{6*w+j}$ || $m_{5*w+j}$ || $m_{4*w+j}$ || $m_{3*w+j}$ || $m_{2*w+j}$ || $m_{w+j}$ || $m_j$
        l ← $i_{7..lwsize}$ || $j_{lwsize-1..0}$
        $a_i$ ← $db_l$
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Wide Translate

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

W.TRANSLATE.8.B	Wide translate bytes big-endian
W.TRANSLATE.16.B	Wide translate doublets big-endian
W.TRANSLATE.32.B	Wide translate quadlets big-endian
W.TRANSLATE.64.B	Wide translate octlets big-endian
W.TRANSLATE.8.L	Wide translate bytes little-endian
W.TRANSLATE.16.L	Wide translate doublets little-endian
W.TRANSLATE.32.L	Wide translate quadlets little-endian
W.TRANSLATE.64.L	Wide translate octlets little-endian

Format

W.TRANSLATE.size.order	rd=rc,rb

rd=wtranslatesizeorder(rc,rb)

31 24	23 18	17 12	11 6	5 4	3 0
W.TRANSLATE.order	rd	rc	rb	sz	0
8	6	6	6	2	4

Description

The contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of a size specified. The low-order bytes of the second group of values are used as addresses to choose entries from one or more tables constructed from the first value, producing a group of values. The group of results is catenated and placed in register rd.

By default, the total width of tables is 128 bits, and a total table width of 128, 64, 32, 16 or 8 bits, but not less than the group size may be specified by adding the desired total table width in bytes to the specified address: 16, 8, 4, 2, or 1. When fewer than 128 bits are specified, the tables repeat to fill the 128 bit width.

The default depth of each table is 256 entries, or in bytes is 32 times the group size in bits. An operation may specify 4, 8, 16, 32, 64, 128 or 256 entry tables, by adding one half of the memory operand size to the address. Table index values are masked to ensure that only the specified portion of the table is used. Tables with just 2 entries cannot be specified; if 2 entry tables are desired, it is recommended to load the entries into registers and use G.MUX to select the table entries.

Failing to initialize the entire table is a potential security hole, as an instruction in with a small-depth table could access table entries previously initialized by an instruction with a large-depth table. We could close this hole either by initializing the entire table, even if extra cycles are required, or by masking the index bits so that only the initialized portion of the table is used. Initializing the entire table with no penalty in cycles could require writing to as many as 128 entries at once, which is quite likely to cause circuit complications. Initializing the entire table with writes to only one entry at a time requires writing 256 cycles, even when the table is smalle:. Masking the index bits is the preferred solution.

Masking the index bits suggests that this instruction, for tables larger than 256 entries, may be useful for a general-purpose memory translate function where the processor performs enough independent load operations to fill the 128 bits. Thus, the 16, 32, and 64 bit versions of this function perform equivalent of 8, 4, 2 withdraw, 8, 4, or 2 load-indexed and 7, 3, or 1 group-extract instructions. In other words, this instruction can be as powerful as 23, 11, or 5 existing instructions. The 8-bit version is a single-cycle operation replacing 47 existing instructions, so these are not as big a win, but nonetheless, this is at least a 50% improvement on a 2-issue processor, even with one-cycle-per load timing. To make this possible, the default table size would become 65536, 2^{32} and 2^{64} for 16, 32 and 64-bit versions of the instruction.

For the big-endian version of this instruction, in the definition below, the contents of register rb is complemented. This reflects a desire to organize the table so that the lowest addressed table entries are selected when the index is zero. In the logical implementation, complementing the index can be avoided by loading the table memory differently f : big-endian and little-endian versions. A consequence of this shortcut is that a table loaded by a big-endian translate instruction cannot be used by a little-endian translate instruction, and vice-versa.

The virtual address must either be aligned to 4096 bytes, or must be the sum of an aligned address and one half of the size of the memory operand in bytes and/or the desired total table width in bytes. An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be a power of two from 4 to 4096 bytes, but must be at least 4 times the group size and 4 times the total table width. If the address is not valid an "access disallowed by virtual address" exception occurs.

Definition

```
def WideTranslate(op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    if $c_{lgsize-4..0}$ ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if $c_{4..lgsize-3}$ ≠ 0 then
        wsize ← (c and (0-c)) || $0^3$
        t ← c and (c-1)
    else
```

```
          wsize ← 128
          t ← c
     endif
     lwsize ← log(wsize)
     if $t_{lwsize+4..lwsize-2} \ne 0$ then
          msize ← (t and (0-t)) || $0^4$
          VirtAddr ← t and (t-1)
     else
          msize ← 256*wsize
          VirtAddr ← t
     endif
     case op of
          W.TRANSLATE.B:
               order ← B
          W.TRANSLATE.L:
               order ← L
     endcase
     m ← LoadMemory(c,VirtAddr,msize,order)
     vsize ← msize/wsize
     lvsize ← log(vsize)
     for i ← 0 to 128-gsize by gsize
          j ← $((order=B)^{lvsize})$ ^ $(b_{lvsize-1+i..i}))$*wsize+$i_{lwsize-1..0}$
          $a_{gsize-1+i..i}$ ← $m_{j+gsize-1..j}$
     endfor
     RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Memory Management

This section discusses the caches, the translation mechanisms, the memory interfaces, and how the multiprocessor interface is used to maintain cache coherence.

Overview

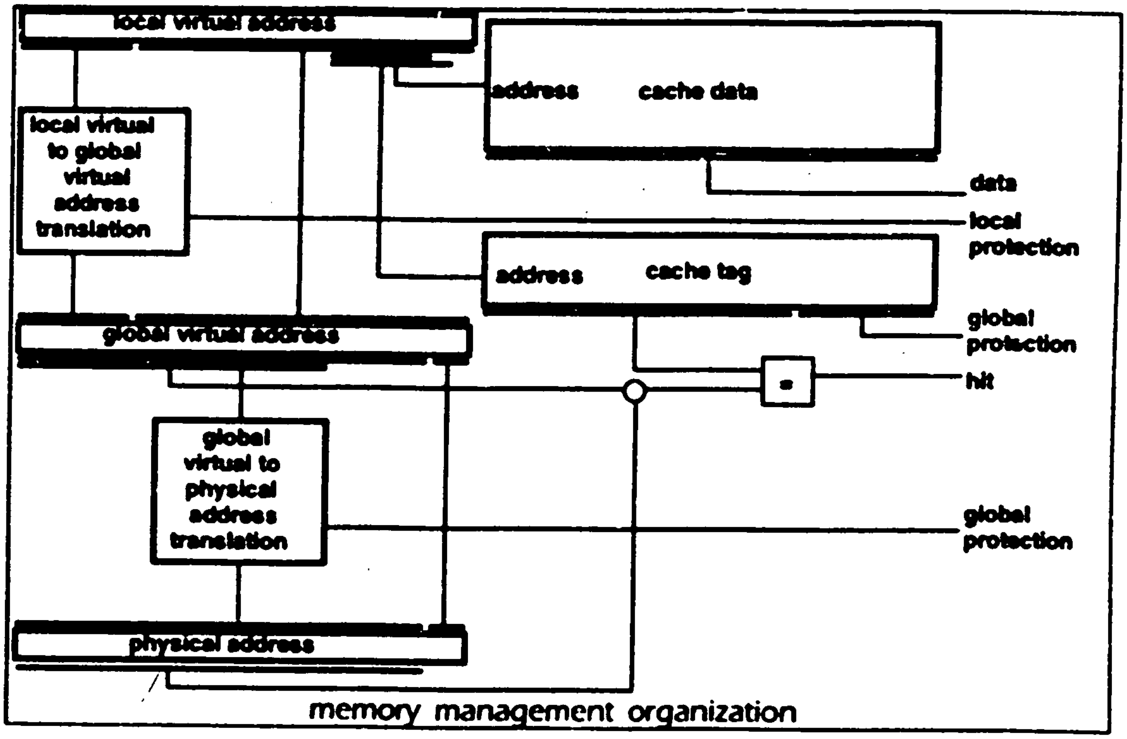
The Zeus processor provides for both local and global virtual addressing, arbitrary page sizes, and coherent-cache multiprocessing. The memory management system is designed to provide the requirements for implementation of virtual machines as well as virtual memory.

All facilities of the memory management system are themselves memory mapped, in order to provide for the manipulation of these facilities by high-level language, compiled code.

The translation mechanism is designed to allow full byte-at-a-time control of access to the virtual address space, with the assistance of fast exception handlers.

Privilege levels provide for the secure transition between insecure user code and secure system facilities. Instructions execute at a privilege, specified by a two-bit field in the access information. Zero is the least-privileged level, and three is the most-privileged level.

The diagram below sketches the basic organization of the memory management system:



memory management organization

In general terms, the memory management starts from a local virtual address. The local virtual address is translated to a global virtual address by a LTB (Local Translation Buffer). In turn, the global virtual address is translated to a physical address by a GTB (Global

Translation Buffer). One of the addresses, a local virtual address, a global virtual address, or a physical address, is used to index the cache data and cache tag arrays, and one of the addresses is used to check the cache tag array for cache presence. Protection information is assembled from the LTB, GTB, and optionally the cache tag, to determine if the access is legal.

This form varies somewhat, depending on implementation choices made. Because the LTB leaves the lower 48 bits of the address alone, indexing of the cache arrays with the local virtual address is usually indentical to cache arrays indexed by the global virtual address. However, indexing cache arrays by the global virtual address rather than the physical address produces a coherence issue if the mapping from global virtual address to physical is many-to-one.

Starting from a local virtual address, the memory management system performs three actions in parallel: the low-order bits of the virtual address are used to directly access the data in the cache, a low-order bit field is used to access the cache tag, and the high-order bits of the virtual address are translated from a local address space to a global virtual address space.

Following these three actions, operations vary depending upon the cache implementation. The cache tag may contain either a physical address and access control information (a physically-tagged cache), or may contain a global virtual address and global protection information (a virtually-tagged cache).

For a physically-tagged cache, the global virtual address is translated to a physical address by the GTB, which generates global protection information. The cache tag is checked against the physical address, to determine a cache hit. In parallel, the local and global protection information is checked.

For a virtually-tagged cache, the cache tag is checked against the global virtual address, to determine a cache hit, and the local and global protection information is checked. If the cache misses, the global virtual address is translated to a physical address by the GTB, which also generates the global protection information.

Local Translation Buffer

The 64-bit global virtual address space is global among all tasks. In a multitask environment, requirements for a task-local address space arise from operations such as the UNIX "fork" function, in which a task is duplicated into parent and child tasks, each now having a unique virtual address space. In addition, when switching tasks, access to one task's address space must be disabled and another task's access enabled.

Zeus provides for portions of the address space to be made local to individual tasks, with a translation to the global virtual space specified by four 16-bit registers for each local virtual space. The registers specify a mask selecting which of the high-order 16 address bits are checked to match a particular value, and if they match, a value with which to modify the virtual address. Zeus avoids setting a fixed page size or local address size; these can be set by software conventions.

A local virtual address space is specified by the following:

field name	size	description
lm	16	mask to select fields of local virtual address to perform match over
la	16	value to perform match with masked local virtual address
lx	16	value to xor with local virtual address if matched
lp	16	local protection field (detailed later)

local virtual address space specifiers

Physical address

There are as many LTB as threads, and up to 2^3 (8) entries per LTB. Each entry is 128 bits, with the high order 64 bits reserved. The physical address of a LTB entry for thread th, entry en, byte b is:

63 … 24	23 … 19	18 … 7	6 … 4	3 … 0
FFFF FFFF 0A00 0000 $_{63..24}$	th	0	en	b
40	5	12	3	4

Definition

```
def data,flags ← AccessPhysicalLTB(pa,op,wdata) as
    th ← $pa_{23..19}$
    en ← $pa_{6..4}$
    if (en < (1 || $0^{LE}$)) and (th < T) and ($pa_{18..6}$=0) then
        case op of
            R:
                data ← $0^{64}$ || LTBArray[th][en]
            W:
                LocalTB[th][en] ← $wdata_{63..0}$
        endcase
    else
        data ← 0
    endif
enddef
```

Entry Format

These 16-bit values are packed together into a 64-bit LTB entry as follows:

63 … 48	47 … 32	31 … 16	15 … 0
lm	la	lx	lp
16	16	16	16

The LTB contains a separate context of register sets for each thread, indicated by the th index above. A context consists of one or more sets of lm/la/lx/lp registers, one set for each simultaneously accessible local virtual address range, indicated by the en index above. This set of registers is called the "Local TB context," or LTB (Local Translation Buffer) context. The effect of this mechanism is to provide the facilities normally attributed to segmentation. However, in this system there is no extension of the address range, instead, segments are local nicknames for portions of the global virtual address space.

A failure to match a LTB entry results either in an exception or an access to the global virtual address space, depending on privilege level. A single bit, selected by the privilege level active for the access from a four bit control register field, global access, ga determines the result. If ga_{PL} is zero (0), the failure causes an exception, if it is one (1), the failure causes the address to be directly used as a global virtual address without modification.

Global Access (fields of control register)

11	10	9	8
ga3	ga2	ga1	ga0
1	1	1	1

Usually, global access is a right conferred to highly privilege levels, so a typical system may be configured with ga0 and ga1 clear (0), but ga2 and ga3 set (1). A single low-privilege (0) task can be safely permitted to have global access, as accesses are further limited by the rwxg privilege fields. A concrete example of this is an emulation task, which may use global addresses to simulate segmentation, such as an x86 emulation. The emulation task then runs as privilege 0, with ga0 set, while most user tasks run as privilege 1, with ga1 clear. Operating system tasks then use privilege 2 and 3 to communicate with and control the user tasks, with ga2 and ga3 set.

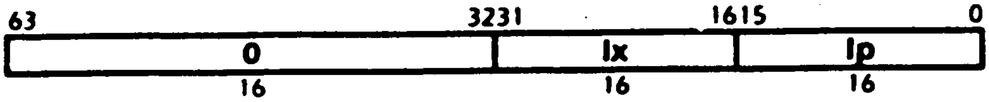
For tasks that have global access disabled at their current privilege level, failure to match a LTB entry causes an exception. The exception handler may load an LTB entry and continue execution, thus providing access to an arbitrary number of local virtual address ranges.

When failure to match a LTB entry does not cause an exception, instructions may access any region in the local virtual address space, when a LTB entry matches, and may access regions in the global virtual address space when no LTB entry matches. This mechanism permits privileged code to make judicious use of local virtual address ranges, which simplifies the manner in which privileged code may manipulate the contents of a local virtual address range on behalf of a less-privileged client. Note, however, that under this model, an LTB miss does not cause an exception directly, so the use of more local virtual address ranges than LTB entries requires more care: the local virtual address ranges should be selected so as not to overlap with the global virtual address ranges, and GTB misses to LVA regions must be detected and cause the handler to load an LTB entry.

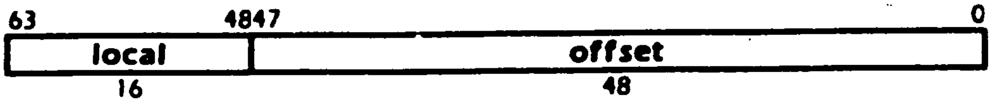
Each thread has an independent LTB, so that threads may independently define local translation. The size of the LTB for each thread is implementation dependent and defined as the LE parameter in the architecture description register. LE is the log of the number of entries in the local TB per thread; an implementation may define LE to be a minimum of 0, meaning one LTB entry per thread, or a maximum of 3, meaning eight LTB entries per thread. For the initial Zeus implementation, each thread has two entries and LE=1.

A minimum implementation of an LTB context is a single set of lm/la/lx/lp registers per thread. However, the need for the LTB to translate both code addresses and data addresses imposes some limits on the use of the LTB in such systems. We need to be able to guarantee forward progress. With a single LTB set per thread, either the code or the data must use global addresses, or both must use the same local address range, as must the LTB and GTB exception handler. To avoid this restriction, the implementation must be raised to two sets per thread, at least one for code and one for data, to guarantee forward progress for arbitrary use of local addresses in the user code (but still be limited to using global addresses for exception handlers).

A single-set LTB context may be further simplified by reserving the implementation of the lm and la registers, setting them to a read-only zero value: Note that in such a configuration, only a single LA region can be implemented.

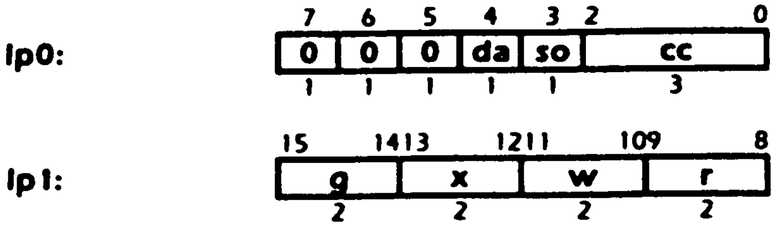


If the largest possible space is reserved for an address space identifier, the virtual address is partitioned as shown below. Any of the bits marked as "local" below may be used as "offset" as desired.



To improve performance, an implementation may perform the LTB translation on the value of the base register (rc) or unincremented program counter, provided that a check is performed which prohibits changing the unmasked upper 16 bits by the add or increment. If this optimization is provided and the check fails, an AccessDisallowedByVirtualAddress should be signaled. If this optimization is provided, the architecture description parameter LB=1. Otherwise LTB translation is performed on the local address, la, no checking is required, and LB=0.

The LTB protect field controls the minimum privilege level required for each memory action of read (r), write (w), execute (x), and gateway (g), as well as memory and cache attributes of write allocate (wa), detail access (da), strong ordering (so), cache disable (cd), and write through (wt). These fields are combined with corresponding bits in the GTB protect field to control these attributes for the mapped memory region.



Field Description

The meaning of the fields are given by the following table:

name	size	meaning
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

Definition

```
def ga,LocalProtect ← LocalTranslation(th,ba,la,pl) as
    if LB & ($ba_{63..48} \ne la_{63..48}$) then
        raise AccessDisallowedByVirtualAddress
    endif
    me ← NONE
    for i ← 0 to (1 || $0^{LE}$)-1
        if ($la_{63..48}$ & ~$LocalTB[th][i]_{63..48}$) = $LocalTB[th][i]_{47..32}$ then
            me ← i
        endif
    endfor
    if me = NONE then
        if ~$ControlRegister_{pl+8}$ then
            raise LocalTBMiss
        endif
        ga ← la
        LocalProtect ← 0
    else
        ga ← ($va_{63..48}$ ^ $LocalTB[th][me]_{31..16}$) || $va_{47..0}$
        LocalProtect ← $LocalTB[th][me]_{15..0}$
    endif
enddef
```

Global Translation Buffer

Global virtual addresses which fail to be accessed in either the LZC, the MTB, the BTB, or PTB are translated to physical references in a table, here named the "Global Translation Buffer," (GTB).

Each processor may have one or more GTB's, with each GTB shared by one or more threads. The parameter GT, the base-two log of the number of threads which share a GTB, and the parameter T, the number of threads, allow computation of the number of GTBs ($T/2^{GT}$), and the number of threads which share each GTB (2^{GT}).

If there are two GTBs and four threads (GT=1, T=4), GTB 0 services references from threads 0 and 1, and GTB 1 services references from threads 2 and 3.

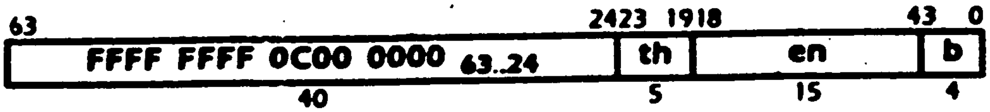
In the first implementation, there is one GTB, shared by all four threads (GT=2, T=4). The GTB has 128 entries (G=7).

Per clock cycle, each GTB can translate one global virtual address to a physical address, yielding protection information as a side effect.

A GTB miss causes a software trap. This trap is designed to permit a fast handler for GlobalTBMiss to be written in software, by permitting a second GTB miss to occur as an exception, rather than a machine check.

Physical address

There may be as many GTB as threads, and up to 2^{15} entries per GTB. The physical address of a GTB entry for thread th, entry en, byte b is:



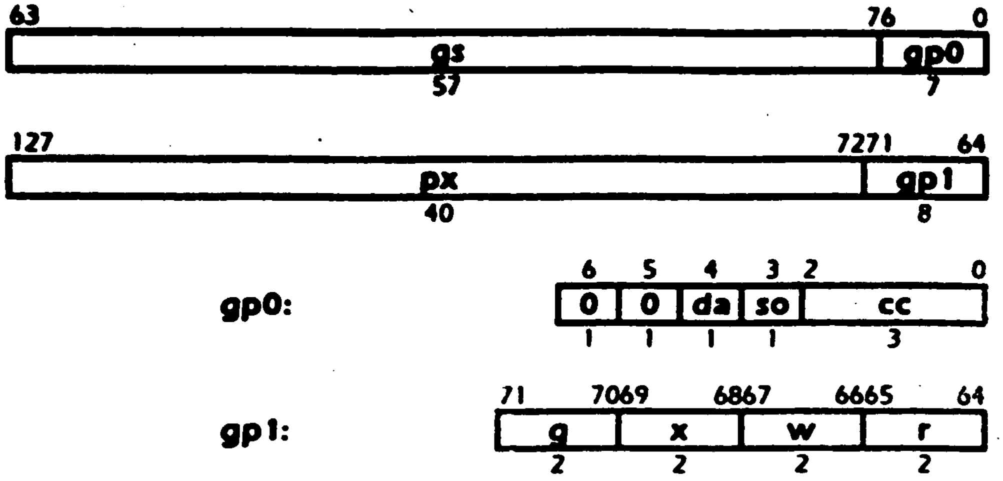
Note that in the diagram above, the low-order GT bits of the th value are ignored, reflecting that 2^{GT} threads share a single GTB. A single GTB shared between threads appears multiple times in the address space. GTB entries are packed together so that entries in a GTB are consecutive:

Definition

```
def data,flags ← AccessPhysicalGTB(pa,op,wdata) as
    th ← $pa_{23..19+GT}$ || $0^{GT}$
    en ← $pa_{18..4}$
    if (en < (1 || $0^{G}$)) and (th < T) and ($pa_{18+GT..19}$ = 0) then
        case op of
            R:
                data ← GTBArray[$th_{5..GT}$][en]
            W:
                GTBArray[$th_{5..GT}$][en] ← wdata
        endcase
    else
        data ← 0
    endif
enddef
```

Entry Format

Each GTB entry is 128 bits. The format of a GTB entry is:



Field Description

gs = ga + size/2: 256 ≤ size ≤ 2^{64}, ga, global address, is aligned (a multiple of) size.

px = pa ^ ga. pa, ga, and px are all aligned (a multiple of) size

The meaning of the fields are given by the following table:

name	size	meaning
gs	57	global address with size
px	56	physical xor
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access
so	1	strong ordering
cc	3	cache control

If the entire contents of the GTB entry is zero (0), the entry will not match any global address at all. If a zero value is written, a zero value is read for the GTB entry. Software must not write a zero value for the gs field unless the entire entry is a zero value.

It is an error to write GTB entries that multiply match any global address; all GTB entries must have unique, non-overlapping coverage of the global address space. Hardware may produce a machine check if such overlapping coverage is detected, or may produce any physical address and protection information and continue execution.

Limiting the GTB entry size to 128 bits allows up to replace entries atomically (with a single store operation), which is less complex than the previous design, in which the mask portion was first reduced, then other entries changed, then the mask is expanded. However, it is limiting the amount of attribute information

or physical address range we can specify. Consequently, we are encoding the size as a single additional bit to the global address in order to allow for attribute information.

Definition

```
def pa,GlobalProtect ← GlobalAddressTranslation(th,ga,pl,lda) as
    me ← NONE
    for i ← 0 to (1 || $0^G$) -1
        if GlobalTB[$th_{5..GT}$][i] ≠ 0 then
            size ← (GlobalTB[$th_{5..GT}$][i]$_{63..7}$ and ($0^{64}$-GlobalTB[$th_{5..GT}$][i]$_{63..7}$)) || $0^8$
            if (($ga_{63..8}$ || $0^8$) ^ (GlobalTB[$th_{5..GT}$][i]$_{63..8}$ || $0^8$)) and ($0^{64}$-size)) = 0 then
                me ← GlobalTB[$th_{5..GT}$][i]
            endif
        endif
    endfor
    if me = NONE then
        if lda then
            PerformAccessDetail(AccessDetailRequiredByLocalTB)
        endif
        raise GlobalTBMiss
    else
        pa ← ($ga_{63..8}$ ^ GlobalTB[$th_{5..GT}$][me]$_{127..72}$) || $ga_{7..0}$
        GlobalProtect ← GlobalTB[$th_{5..GT}$][me]$_{71..64}$ || $0^1$ || GlobalTB[$th_{5..GT}$][me]$_{6..0}$
    endif
enddef
```

GTB Registers

Because the processor contains multiple threads of execution, even when taking virtual memory exceptions, it is possible for two threads to nearly simultaneously invoke software GTB miss exception handlers for the same memory region. In order to avoid producing improper GTB state in such cases, the GTB includes access facilities for indivisibly checking and then updating the contents of the GTB as a result of a memory write to specific addresses.

A 128-bit write to the address GTBUpdateFill (fill=1), as a side effect, causes first a check of the global address specified in the data against the GTB. If the global address check results in a match, the data is directed to write on the matching entry. If there is no match, the address specified by GTBLast is used, and GTBLast is incremented. If incrementing GTBLast results in a zero value, GTBLast is reset to GTBFirst, and GTBBump is set. Note that if the size of the updated value is not equal to the size of the matching entry, the global address check may not adequately ensure that no other entries also cover the address range of the updated value. The operation is unpredictable if multiple entries match the global address.

/

The GTBUpdateFill register is a 128-bit memory-mapped location, to which a write operation performes the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry.

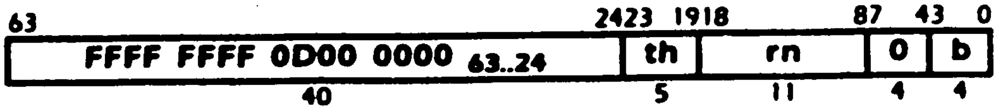
An alternative write address, GTBUpdate, (fill=0) updates a matching entry, but makes no change to the GTB if no entry matches. This operation can be used to indivisibly update a GTB entry as to protection or physical address information.

Definition

```
def GTBUpdateWrite(th,fill,data) as
    me ← NONE
    for i ← 0 to (1 || $0^G$) -1
        size ← (GlobalTB[$th_{5..GT}$][i]$_{63..7}$ and ($0^{64}$-GlobalTB[$th_{5..GT}$][i]$_{63..7}$)) || $0^8$
        if (($data_{63..8}$||$0^8$) ^ (GlobalTB[$th_{5..GT}$][i]$_{63..8}$||$0^8$)) and ($0^{64}$-size) = 0 then
            me ← i
        endif
    endfor
    if me = NONE then
        if fill then
            GlobalTB[$th_{5..GT}$][GTBLast[$th_{5..GT}$]] ← data
            GTBLast[$th_{5..GT}$] ← (GTBLast[$th_{5..GT}$] + 1)$_{G-1..0}$
            if GTBLast[$th_{5..GT}$] = 0 then
                GTBLast[$th_{5..GT}$] ← GTBFirst[$th_{5..GT}$]
                GTBBump[$th_{5..GT}$] ← 1
            endif
        endif
    else
        GlobalTB[$th_{5..GT}$][me] ← data
    endif
enddef
```

Physical address

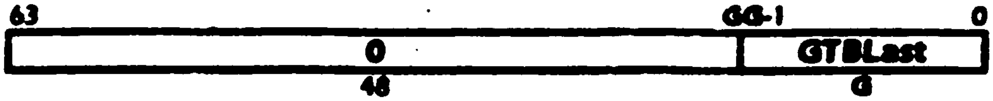
There may be as many GTB as threads, and up to 2^{11} registers per GTB (5 registers are implemented). The physical address of a GTB control register for thread th, register rn, byte b is:

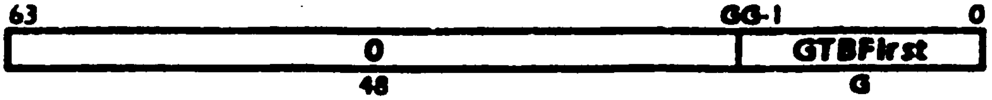


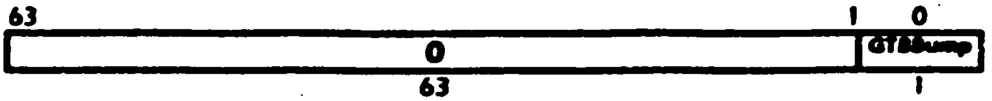
Note that in the diagram above, the low-order GT bits of the th value are ignored, reflecting that 2^{GT} threads share single GTB registers. A single set of GTB registers shared between threads appears multiple times in the address space, and manipulates the GTB of the threads with which the registers are associated.

The GTBUpdate register is a 128-bit memory-mapped location, to which a write operation performes the operation defined above. A read operation returns a zero value. The format of the GTBUpdateFill register is identical to that of a GTB entry.

The registers GTBLast, GTBFirst, and GTBBump are memory mapped. The GTBLast and GTBFirst registers are G bits wide, and the GTBBump register is one bit.:







Definition

```
def data,flags ← AccessPhysicalGTBRegisters(pa,op,wdata) as
    th ← $pa_{23..19+GT}$ || $0^{GT}$
    m ← $pa_{18..8}$
    if (m < 5) and (th < T) and ($pa_{18+GT..19}$ = 0) and ($pa_{7..4}$ = 0) then
        case m || op of
            0 || R, 1 || R:
                data ← 0
            0 || W, 1 || W:
                GTBUpdateWrite(th,$m_0$,wdata)
            2 || R:
                data ← $0^{64-G}$ || GTBLast[$th_{5..GT}$]
            2 || W:
                GTBLast[$th_{5..GT}$] ← $wdata_{G-1..0}$
            3 || R:
                data ← $0^{64-G}$ || GTBFirst[$th_{5..GT}$]
            3 || W:
                GTBFirst[$th_{5..GT}$] ← $wdata_{G-1..0}$
            3 || R:
                data ← $0^{63}$ || GTBBump[$th_{5..GT}$]
            3 || W:
                GTBBump[$th_{5..GT}$] ← $wdata_0$
        endcase
    else
        data ← 0
    endif
enddef
```

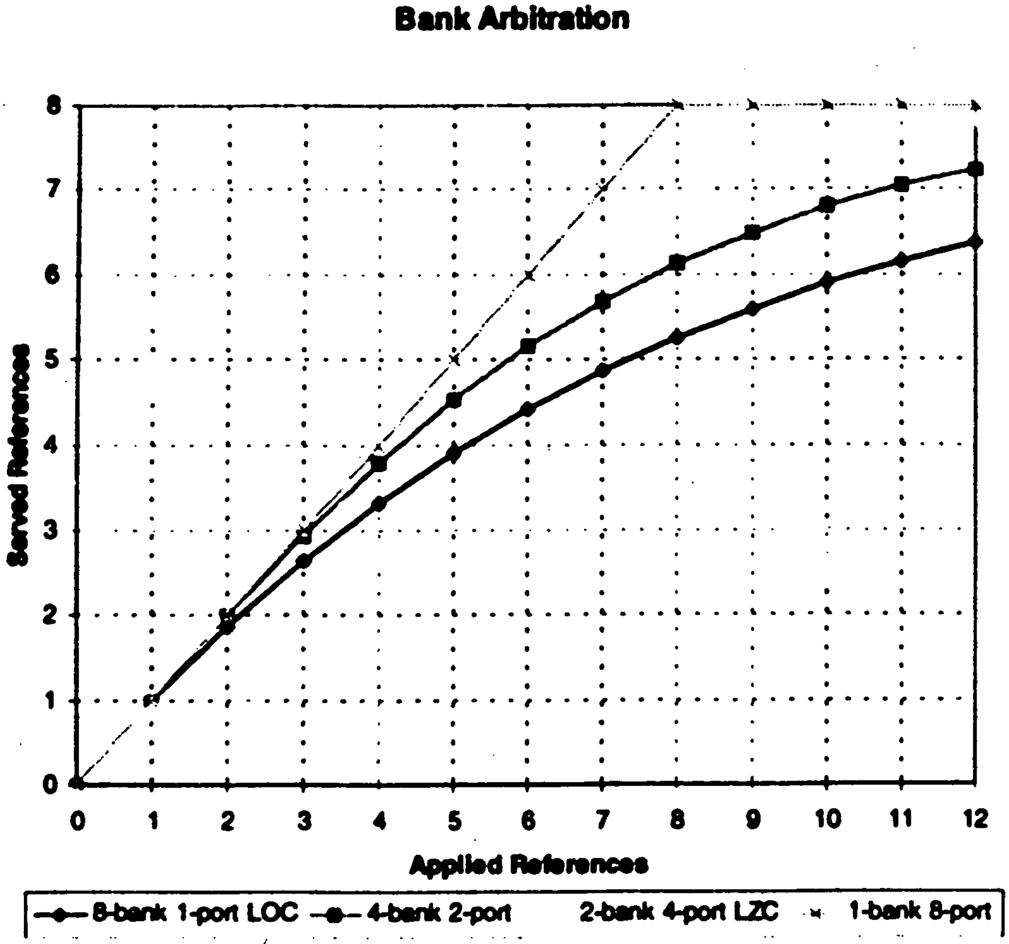
Address Generation

The address units of each of the four threads pr ide up to two global virtual addresses of load, store, or memory instructions, for a total of eight addresses. LTB units associated with each thread translate the local addresses into global addresses. The LZC operates on global addresses. MTB, BTB, and PTB units associated with each thread translate the global addresses into physical addresses and cache addresses. (A PTB unit associated with each thread produces physical addresses and cache addresses for program counter references. – this is optional, as by limiting address generation to two per thread, the MTB can be used for program references.) Cache addresses are presented to the LOC as required, and physical addresses are checked against cache tags as required.

Memory Banks

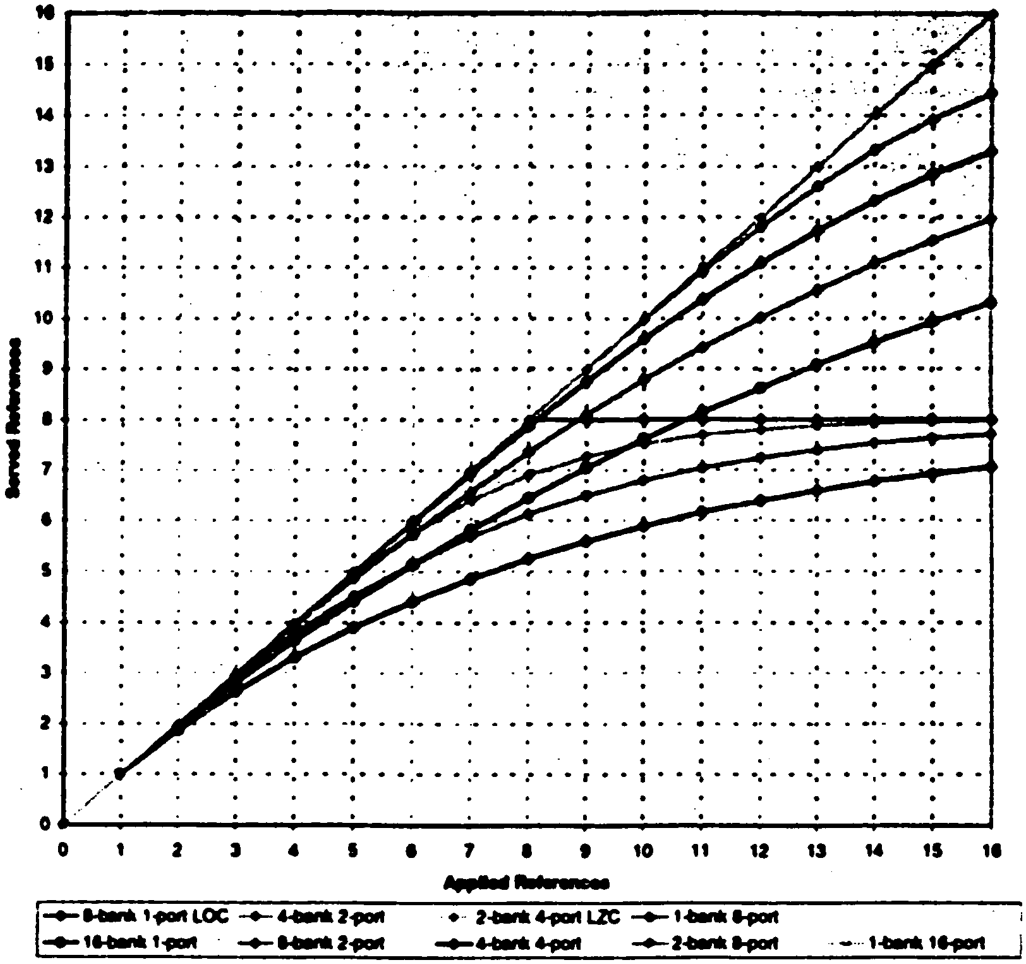
The LZC has two banks, each servicing up to four requests. The LOC has eight banks, each servicing at most one request.

Assuming random request addresses, the following graph shows the expected rate at which requests are serviced by multi-bank/multi-port memories that have 8 total ports and divided into 1, 2, 4, or 8 interleaved banks. The LZC is 2 bank·, each with 4 ports, and the LOC is 8 banks, each 1 port.



Note a small difference between applying 12 references versus 8 references for the LOC (6.5 vs 5.2), and for the LZC (7.8 vs. 6.9). This suggests that simplifying the system to produce two address per thread (program+load/store or two load/store) will not overly hurt performance. A closer simulation, taking into account the sequential nature of the program and load/store traffic may well yield better numbers, as threads will tend to line up in non-interfering patterns, and program microcaching reduces program fetching.

The following graph shows the rates for both 8 total ports and 16 total ports.



Note significant differences between 8-port systems and 16-port systems, even when used with a maximum of 8 applied references. In particular, a 16-bank 1-port system is better than a 4-bank 2-port system with more than 6 applied references. Current layout estimates would require about a 14% area increase (assuming no savings from smaller/simpler sense amps) to switch to a 16-port LOC, with a 22% increase in 8-reference throughput.

Program Microcache

A program microcache (PMC) which holds only program code for each thread may optionally exist, and does exist for the initial implementation. The program microcache is flushed by reset, or by executing a B.BARRIER instruction. The program microcache is always clean, and is not snooped by writes or otherwise kept coherent, except by flushing as indicated above. The microcache is not altered by writing to the LTB or GTB, and software must execute a B.BARRIER instruction before expecting the new contents of tl . LTB or GTB to affect determination of PMC hit or miss status on program fetches.

In the initial implementation, the program microcache holds simple loop code. The microcache holds two separately addressed cache lines. Branches or execution beyond this region cause the microcache to be flushed and refilled at the new address, provided that the addresses are executable by the current thread. The program microcache uses the B.HINT and B.HINT.I to accelerate fetching of program code when possible. The program microcache generally functions as a prefetch buffer, except that short forward or backward branches within the region covered maintain the contents of the microcache.

Program fetches into the microcache are requested on any cycle in which less than two load/store addresses are generated by the address unit, unless the microcache is already full. System arbitration logic should give program fetches lower priority than load/store references when first presented, then equal priority if the fetch fails arbitration a certain number of times. The delay until program fetches have equal priority should be based on the expected time the program fetch data will be executed; it may be as small as a single cycle, or greater for fetches which are far ahead of the execution point.

Wide Microcache

A wide microcache (WMC) which holds only data fetched for wide (W) instructions may optionally exist, and does exist for the initial implementation, for each unit which implements one or more wide (W) instructions.

The wide (W) instructions each operate on a block of data fetched from memory and the contents of one or more registers, producing a result in a register. Generally, the amount of data in the block exceeds the maximum amount of data that the memory system can supply in a single cycle, so caching the memory data is of particular importance. All the wide (W) instructions require that the memory data be located at an aligned address, an address that is a multiple of the size of the memory data, which is always a power of two.

The wide (W) instructions are performed by functional units which normally perform execute or "back-end" instructions, though the loading of the memory data requires use of the access or "front-end" functional units. To minimize the use of the "front-end" functional units, special rules are used to maintain the coherence of a wide microcache (WMC).

Execution of a wide (W) instruction has a residual effect of loading the specified memory data into a wide microcache (WMC). Under certain conditions, a future wide (W) instruction may be able to reuse the WMC contents.

First of all, any store or cache coherency action on the physical addresses referenced by the WMC will invalidate the contents. The minimum translation unit of the virtual memory system, 256 bytes, defines the number of physical address blocks which must be checked by any store. A WMC for the W.TABLE instruction may be as large as 4096 bytes, and so requires as many as 16 such physical address blocks to be checked for each WMC entry. A WMC for the W.SWITCH or W.MUL.* instructions need check only one address block for each WMC entry, as the maximum size is 128 bytes.

By making these checks on the physical addresses, we do not need to be concerned about changes to the virtual memory mapping from virtual to physical addresses, and the virtual memory state can be freely changed without invalidating any WMC.

Absent any of the above changes, the WMC is only valid if it contains the contents relevant to the current wide (W) instruction. To check this with minimal use of the front-end units, each WMC entry contains a first tag with the thread and address register for which it was last used. If the current wide (W) instruction uses the same thread and address register, it may proceed safely. Any intervening writes to that address register by that thread invalidates the WMC thread and address register tag.

If the above test fails, the front-end is used to fetch the address register and check its contents against a second WMC tag, with the physical addresses for which it was last used. If the tag matches, it may proceed safely. As detailed above, any intervening stores or cache coherency action by any thread to the physical addresses invalidates the WMC entry.

If both the above tests fail for all relevant WMC entries, there is no alternative but to load the data from the virtual memory system into the WMC. The front-end units are responsible for generating the necessary addresses to the virtual memory system to fetch the entire data block into a WMC.

For the first implementation, it is anticipated that there be eight WMC entries for each of the two X units (for W.SWITCH instructions), eight WMC entries for each of the two E units (for W.MUL instructions), and four WMC entries for the single T unit. The total number of WMC address tags requires is 8*2*1+8*2*1+4*1*16 = 96 entries.

The number of WMC address tags can be substantially reduced to 32+4=36 entries by making an implementation restriction requiring that a single translation block be used to translate the data address of W.TABLE instructions. With this restriction, each W.TABLE WMC entry uses a contiguous and aligned physical data memory block, for which a single address tag can contain the relevant information. The size of such a block is a maximum of 4096 bytes. The restriction can be checked by examining the size field of the referenced GTB entry.

Level Zero Cache

The innermost cache level, here named the "Level Zero Cache," (LZC) is fully associative and indexed by global address. Entries in the LZC contain global addresses and previously fetched data from the memory system. The LZC is an implementation feature, not visible to the Zeus architecture.

Entries in the LZC are also used to hold the global addresses of store instructions that have been issued, but not yet completed in the memory system. The LZC entry may also contain the data associated with the global address, as maintained either before or after updating with the store data. When it contains the post-store data, results of stores may be forwarded directly to the requested reference.

With an LZC hit, data is returned from the LZC data, and protection from the LZC tag. No LOC access is required to complete the reference.

All loads and program fetches are checked against the LZC for conflicts with entries being used as store buffer. On a LZC hit on such entries, if the post-store data is present, data may be returned by the LZC to satisfy the load or program fetch. If the post-store data is not present, the load or program fetch must stall until the data is available.

With an LZC miss, a victim entry is selected, and if dirty, the victim entry is written to the LOC. The LOC cache is accessed, and a valid LZC entry is constructed from data from the LOC and tags from the LOC protection information.

All stores are checked against the LZC for conflicts, and further cause a new entry in the LZC, or "take over" a previously clean LZC entry for this purpose. Unaligned stores may require two entries in the LZC. At time of allocation, the address is filled in.

Two operations then occur in parallel - 1) for write-back cached references, the remaining bytes of the hexlet are loaded from the LOC (or LZC), and 2) the addressed bytes are filled in with data from data path. If an exception causes the store to be purged before retirement, the LZC entry is marked invalid, and not written back. When the store is retired, the LZC entry can be written back to LOC or external interface.

Structure

The eight memory addresses are partitioned into up to four odd addresses, and four even addresses.

The LZC contains 16 fully associative entries that may each contain a single hexlet of data at even hexlet addresses (LZCE), and another 16 entries for odd hexlet addresses (LZCO). The maximum capacity of the LZC is 16*32=512 bytes.

The tags for these entries are indexed by global virtual address (63..5), and contain access control information, detailed below.

The address of entries accessed associatively is also encoded into binary and provided as output from the tags for use in updating the LZC, through its write ports.

8 bit rwxg

16 bit valid

16 bit dirty

4 bit LO$ address

16 bit protection

```
def data,protect,valid,dirty,match ← LevelZeroCacheRead(ga) as
    eo ← $ga_4$
    match ← NONE
    for i ← 0 to LevelZeroCacheEntries/2-1
        if ($ga_{63..5}$ = LevelZeroTag[eo][i]) then
            match ← i
        endif
    endfor
    if match = NONE then
        raise LevelZeroCacheMiss
    else
        data ← $LevelZeroData[eo][match]_{127..0}$
        valid ← $LevelZeroData[eo][match]_{143..128}$
        dirty ← $LevelZeroData[eo][match]_{159..144}$
        protect ← $LevelZeroData[eo][match]_{167..160}$
    endif,
enddef
```

Level One Cache

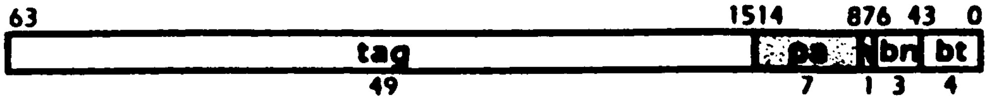
The next cache level, here named the "Level One Cache," (LOC) is four-set-associative and indexed by the physical address. The eight memory addresses are partitioned into up to eight addresses for each of eight independent memory banks. The LOC has a cache block size of 256 bytes, with triclet (32-byte) sub-blocks.

The LOC may be partitioned into two sections, one part used as a cache, and the remainder used as "niche memory." Niche memory is at least as fast as cache memory, but unlike cache, never misses to main memory. Niche memory may be placed at any virtual address, and has physical addresses fixed in the memory map. The **nl** field in the **control register** configures the partitioning of LOC into cache memory and niche memory.

The LOC data memory is (256+8)x4x(128+2) bits, depth to hold 256 entries in each of four sets, each entry consisting of one hexlet of data (128 bits), one bit of parity, and one spare bit. The additional 8 entries in each of four sets hold the LOC tags, with 128 bits per entry for 1/8 of the total cache, using 512 bytes per data memory and 4K bytes total.

There are 128 cache blocks per set, or 512 cache blocks total. The maximum capacity of the LOC is 128k bytes. Used as a cache, the LOC is partitioned into 4 sets, each 32k bytes. Physically, the LOC is partitioned into 8 interleaved physical blocks, each holding 16k bytes.

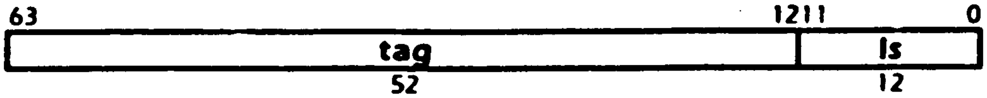
The physical address **$pa_{63..0}$** is partitioned as below into a 52 to 54 bit **tag** (three to five bits are duplicated from the following field to accommodate use of portion of the cache as niche), 8-bit address to the memory bank (7 bits are physical address (**pa**), 1 bit is virtual address (**v**)), 3 bit memory bank select (**bn**), and 4-bit byte address (**bt**). All access to the LOC are in units of 128 bits (hexlets), so the 4-bit byte address (**bt**) does not apply here. The shaded field (**pa,v**) is translated via **nl** to a cache identifier (**ci**) and set identifier (**si**) and presented to the LOC as the LOC address to LOC bank **bn**.

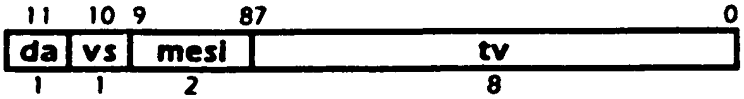


The LOC tag consists of 64 bits of information, including a 52 to 54-bit tag and other cache state information. Only one MTB entry at a time may contain a LOC tag.

With 256 byte cache lines, there are 512 cache blocks. At 64 bits per tag, the cache tags require 4k bytes of storage. This storage is adjacent to the LOC data memory itself, using physical addresses = 1024..1055. Alternatively (see detailed description below), physical addresses = 0..31 may be used.

The format of a LOC tag entry is shown below.



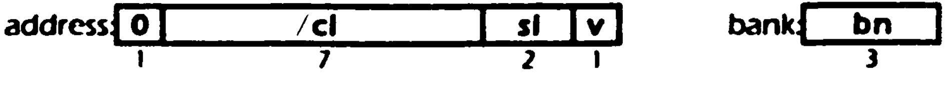


The meaning of the fields are given by the following table:

name	size	meaning
tag	52	physical address tag
da	1	detail access (or physical address bit 11)
vs	1	victim select (or physical address bit 10)
mesi	2	coherency: modified (3), exclusive (2), shared (1), invalid (0)
tv	8	triclet valid (1) or invalid (0)

To access the LOC, a global address is supplied to the Micro-Tag Buffer (MTB), which associatively looks up the global address into a table holding a subset of the LOC tags. In particular, each MTB table entry contains the cache index derived from physical address bits 14..8, **ci**, (7 bits) and set identifier, **si**, (2 bits) required to access the LOC data. Each MTB table entry also contains the protection information of the LOC tag.

With an MTB hit, protection information is supplied from the MTB. The MTB supplies the resulting cache index (ci, from the MTB), set identifier, **si**, (2 bits) and virtual address (bit 7, **v**, from the LA), which are applied to the LOC data bank selected from bits 6..4 of the LA. The diagram below shows the address presented to LOC data bank **bn**.



With an MTB miss, the GTB (described below) is referenced to obtain a physical address and protection information.

To select the cache line, a 7-bit niche limit register **nl** is compared against the value of **pa14..8** from the GTB. If **pa14..8<nl**, a 7-bit address modifier register **am** is inclusive-or'ed against **pa14..8**, producing a cache index, **ci**. Otherwise, **pa14..8** is used as **ci**. Cache lines 0..**nl**-1, and cache tags 0..**nl**-1, are available for use as niche memory. Cache lines **nl**..127 and cache tags **nl**..127 are used as LOC.

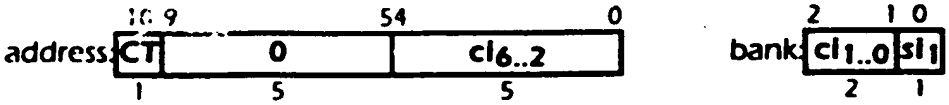
$$ci \leftarrow (pa_{14..8} < nl)\ ?\ (pa_{14..8} \mid\mid am)\ :\ pa_{14..8}$$

The address modifier **am** is $(1^{7-\log(128-nl)} \mid\mid 0^{\log(128-nl)})$. The **bt** field specifies the least-significant bit used for tag, and is (**nl**<112) ? 12 : 8+log(128-**nl**):

nl	am	bt
0	0	12
1..64	64	12
65..96	96	12
97..112	112	12
113..120	120	11
121..124	124	10
125..126	126	9
127	127	8

Values for **nl** in the range 113..127 require more than 52 physical address tag bits in the LOC tag and a requisite reduction in LOC features. Note that the presence of bits 14..10 of the physical address in the LOC tag is a result of the possibility that, with am=64..127, the cache index value **ci** cannot be relied upon to supply bit 14..8. Bits 9..8 can be safely inferred from the cache index value **ci**, so long as **nl** is in the range 0..124. When **nl** is in the range 113..127, the **da** bit is used for bit 11 of the physical address, so the Tag detail access bit is suppressed. When **nl** is in the range 121..127, the **vs** bit is used for bit 10 of the physical address, so victim selection is performed without state bits in the LOC tag. When **nl** is in the range 125..127, the set associativity is decreased, so that **si1** is used for bit 9 of the physical address and when nl is 127, **si0** is used for bit 8 of the physical address.

Four tags are fetched from the LOC tags and compared against the PA to determine which of the four sets contain the data. The four tags are contained in two consecutive banks; they may be simultaneously or independently fetched. The diagram below shows the address presented to LOC data bank (**ci1..0** || **si1**).



Note that the **CT** architecture description variable is present in the above address. **CT** describes whether dedicated locations exist in the LOC for tags at the next power-of-two boundary above the LOC data. The niche-mapping mechanism can provide the storage for the LOC tags, so the existence of these dedicated tags is optional: If **CT**=0, addresses at the beginning of the LOC (0..31 for this implementation) are used for LOC tags, and the **nl** value should be adjusted accordingly by software.

The LOC address (ci || si) uniquely identifies the cache location, and this LOC address is associatively checked against all MTB entries on changes to the LOC tags, such as by cache block replacement, bus snooping, or software modification. Any matching MTB entries are flushed, even if the MTB entry specifies a different global address - this permits address aliasing (the use of a physical address with more than one global address.

With an LOC miss, a victim set is selected (LOC victim selection is described below), whose contents, if any sub-block is modified, is written to the external memory. A new LOC entry is constructed with address and protection information from the GTB, and data fetched from external memory.

The diagram below shows the contents of LOC data memory banks 0..7 for addresses 0..2047:

address	bank 7	...	bank 1	bank 0
0	line 0, hexlet 7, set 0		line 0, hexlet 1, set 0	line 0, hexlet 0, set 0
1	line 0, hexlet 15, set 0		line 0, hexlet 9, set 0	line 0, hexlet 8, set 0
2	line 0, hexlet 7, set 1		line 0, hexlet 1, set 1	line 0, hexlet 0, set 1
3	line 0, hexlet 15, set 1		line 0, hexlet 9, set 1	line 0, hexlet 8, set 1
4	line 0, hexlet 7, set 2		line 0, hexlet 1, set 2	line 0, hexlet 0, set 2
5	line 0, hexlet 15, set 2		line 0, hexlet 9, set 2	line 0, hexlet 8, set 2
6	line 0, hexlet 7, set 3		line 0, hexlet 1, set 3	line 0, hexlet 0, set 3
7	line 0, hexlet 15, set 3		line 0, hexlet 9, set 3	line 0, hexlet 8, set 3
8	line 1, hexlet 7, set 0		line 1, hexlet 1, set 0	line 1, hexlet 0, set 0
9	line 1, hexlet 15, set 0		line 1, hexlet 9, set 0	line 1, hexlet 8, set 0
10	line 1, hexlet 7, set 1		line 1, hexlet 1, set 1	line 1, hexlet 0, set 1
11	line 1, hexlet 15, set 1		line 1, hexlet 9, set 1	line 1, hexlet 8, set 1
12	line 1, hexlet 7, set 2		line 1, hexlet 1, set 2	line 1, hexlet 0, set 2
13	line 1, hexlet 15, set 2		line 1, hexlet 9, set 2	line 1, hexlet 8, set 2
14	line 1, hexlet 7, set 3		line 1, hexlet 1, set 3	line 1, hexlet 0, set 3
15	line 1, hexlet 15, set 3		line 1, hexlet 9, set 3	line 1, hexlet 8, set 3
...	...		...	...
1016	line 127, hexlet 7, set 0		line 127, hexlet 1, set 0	line 127, hexlet 0, set 0
1017	line 127, hexlet 15, set 0		line 127, hexlet 9, set 0	line 127, hexlet 8, set 0
1018	line 127, hexlet 7, set 1		line 127, hexlet 1, set 1	line 127, hexlet 0, set 1
1019	line 127, hexlet 15, set 1		line 127, hexlet 9, set 1	line 127, hexlet 8, set 1
1020	line 127, hexlet 7, set 2		line 127, hexlet 1, set 2	line 127, hexlet 0, set 2
1021	line 127, hexlet 15, set 2		line 127, hexlet 9, set 2	line 127, hexlet 8, set 2
1022	line 127, hexlet 7, set 3		line 127, hexlet 1, set 3	line 127, hexlet 0, set 3
1023	line 127, hexlet 15, set 3		line 127, hexlet 9, set 3	line 127, hexlet 8, set 3
1024	tag line 3, sets 3 and 2		tag line 0, sets 3 and 2	tag line 0, sets 1 and 0
1025	tag line 7, sets 3 and 2		tag line 4, sets 3 and 2	tag line 4, sets 1 and 0
...	...		...	...
1055	tag line 127, sets 3 and 2		tag line 124, sets 3 and 2	tag line 124, sets 1 and 0
1056	reserved		reserved	reserved
...	...		...	...
2047	reserved		reserved	reserved

The following table summarizes the state transitions required by the LOC cache:

cc	op	mesi	v	bus op	c	x	mesi	v	w	m	notes
NC	R	x	x	uncached read							
NC	W	x	x	uncached write							
CD	R	I	x	uncached read							
CD	R	x	0	uncached read							
CD	R	MES	1	(hit)							
CD	W	I	x	uncached write							
CD	W	x	0	uncached write							
CD	W	MES	1	uncached write						1	
WT/WA	R	I	x	triclet read	0	x					
WT/WA	R	I	x	triclet read	1	0	S	1			
WT/WA	R	I	x	triclet read	1	1	E	1			
WT/WA	R	MES	0	triclet read	0	x					inconsistent KEN#
WT/WA	R	S	0	triclet read	1	0		1			
WT/WA	R	S	0	triclet read	1	1		1			E->S: extra sharing
WT/WA	R	E	0	triclet read	1	0		1			
WT/WA	R	E	0	triclet read	1	1	S	1			shared block
WT/WA	R	M	0	triclet read	1	0	S	1			other subblocks M->I
WT/WA	R	M	0	triclet read	1	1		1			E->M: extra dirty
WT/WA	R	MES	1	(hit)							
WT	W	I	x	uncached write							
WT	W	x	0	uncached write							
WT	W	MES	1	uncached write						1	
WA	W	I	x	triclet read	0	x			1		throwaway read
WA	W	I	x	triclet read	1	0	S	1	1	1	
WA	W	I	x	triclet read	1	1	M	1		1	
WA	W	MES	0	triclet read	0	x			1	1	inconsistent KEN#
WA	W	S	0	triclet read	1	0	S	1	1	1	
WA	W	S	0	triclet read	1	1	M	1		1	
WA	W	S	1	write		0	S	1		1	
WA	W	S	1	write		1	S	1		1	E->S: extra sharing
WA	W	E	0	triclet read	1	0	S	1	1	1	
WA	W	E	0	triclet read	1	1	E	1	1	1	
WA	W	E	1	(hit)		x	M	1			E->M: extra dirty
WA	W	M	0	triclet read	1	0	M	1	1	1	
WA	W	M	0	triclet read	1	1	M	1		1	
WA	W	M	1	(hit)		x	M	1			

cc	cache control
op	operation: R=read, W=write
mesi	current mesi state
v	current tv state
bus op	bus operation
c	cachable (triclet) result
x	exclusive result
mesi	new mesi state
v	new tv state
w	cacheable write after read
m	merge store data with cache line data
notes	other notes on transition

Definition

```
def data,tda ← LevelOneCacheAccess(pa,size,lda,gda,cc,op,wd) as

    // cache index
    am ← ($1^{7-\log(128-nl)}$ || $0^{\log(128-nl)}$)
    ci ← ($pa_{14..8}$<nl) ? ($pa_{14..8}$ || am) : $pa_{14..8}$
    bt ← (nl≤112) ? 12 : 8+log(128-nl)

    // fetch tags for all four sets
    tag10 ← ReadPhysical(0xFFFFFFFF00000000$_{63..19}$ || CT || $0^5$ || ci || $0^1$ || $0^4$, 128)
    Tag[0] ← $tag10_{63..0}$
    Tag[1] ← $tag10_{127..64}$
    tag32 ← ReadPhysical(0xFFFFFFFF00000000$_{63..19}$ || CT || $0^5$ || ci || $1^1$ || $0^4$, 128)
    Tag[2] ← $tag32_{63..0}$
    Tag[3] ← $tag32_{127..64}$
    vsc ← ($Tag[3]_{10}$ || $Tag[2]_{10}$) ^ ($Tag[1]_{10}$ || $Tag[0]_{10}$)

    // look for matching tag
    si ← MISS
    for i ← 0 to 3
        if ($Tag[i]_{63..10}$ || $i_{1..0}$ || $0^7$)$_{63..bt}$ = $pa_{63..bt}$ then
            si ← i
        endif
    endfor

    // detail access checking on MISS
    if (si = MISS) and (lda ≠ gda) then
        if gda then
            PerformAccessDetail(AccessDetailRequiredByGlobalTB)
        else
            PerformAccessDetail(AccessDetailRequiredByLocalTB)
        endif
    endif

    // if no matching tag or invalid MESI or no sub-block, perform cacheable read/write
    bd ← (si = MISS) or ($Tag[si]_{9..8}$ = I) or ((op=W) and ($Tag[si]_{9..8}$ = S)) or ~$Tag[si]_{pa_{7..5}}$
    if bd then
```

```
          if (op=W) and (cc ≥ WA) and ((si = MISS) or ~$Tag[si]_{pa_{7..5}}$ or ($Tag[si]_{9..8}$ ≠ S)) then
                    data,cen,xen ← AccessPhysical(pa,size,cc,R,0)
                    //if cache disabled or shared, do a write through
                    if ~cen or ~xen then
                              data,cen,xen ← AccessPhysical(pa,size,cc,W,wd)
                    endif
          else
                    data,cen,xen ← AccessPhysical(pa,size,cc,op,wd)
          endif
          al ← cen
else
          al ← 0
endif

// find victim set and eject from cache
if al and (si = MISS or $Tag[si]_{9..8}$ = I) then
          case bt of
                    12..11:
                              si ← vsc
                    10..8:
                              gvsc ← gvsc + 1
                              si ← (bt≤9) : $pa_9$ : $gvsc_1$^$pa_{11}$ || (bt≤8) : $pa_8$ : $gvsc_0$^$pa_{10}$
          endcase
          if $Tag[si]_{9..8}$ = M then
                    for i ← 0 to 7
                              if $Tag[si]_i$ then
                                        vca ← $0xFFFFFFFF00000000_{63..19}$ || 0 || ci || si || $i_{2..0}$ || $0^4$
                                        vdata ← ReadPhysical(vca, 256)
                                        vpa ← ($Tag[si]_{63..10}$ || $si_{1..0}$ || $0^7)_{63..bt}$ || $pa_{bt-1..8}$ || $i_{2..0}$ || 0 || $0^4$
                                        WritePhysical(vpa, 256, vdata)
                              endif
                    endfor
          endif
          if $Tag[vsc+1]_{9..8}$ = I then
                    nvsc ← vsc + 1
          elseif $Tag[vsc+2]_{9..8}$ = I then
                    nvsc ← vsc + 2
          elseif $Tag[vsc+3]_{9..8}$ = I then
                    nvsc ← vsc + 3
          else
                    case cc of
                              NC, CD, WT, WA, PF:
                                        nvsc ← vsc + 1
                              LS, SS:
                                        nvsc ← vsc //no change
                              endif
                    endcase
          endif
          tda ← 0
          sm ← $0^{7-pa_{7..5}}$ || $1^1$ || $0^{pa_{7..5}}$
else
          nvsc ← vsc
          tda ← (bt>11) ? $Tag[si]_{11}$ : 0
          if al then
```

```
            sm ← $Tag[si]_{7..1+pa_{7..5}}$ || $1^1$ || $Tag[si]_{pa_{7..5}-1..0}$
        endif
    endif

    // write new data into cache and update victim selection and other tag fields
    if al then
        if op=R then
            mesi ← xen ? E : S
        else
            mesi ← xen ? M : I TODO
        endif
        case bt of
            12:
                Tag[si] ← $pa_{63..bt}$ || tda || $Tag[si\hat{}2]_{10}$ ^ $nvsc_{si_0}$ || mesi || sm
                $Tag[si\hat{}1]_{10}$ ← $Tag[si\hat{}3]_{10}$ ^ $nvsc_{1\hat{}si_0}$
            11:
                Tag[si] ← $pa_{63..bt}$ || $Tag[si\hat{}2]_{10}$ ^ $nvsc_{si_0}$ || mesi || sm
                $Tag[si\hat{}1]_{10}$ ← $Tag[si\hat{}3]_{10}$ ^ $nvsc_{1\hat{}si_0}$
            10:
                Tag[si] ← $pa_{63..bt}$ || mesi || sm
        endcase
        dt ← 1
        nca ← $0xFFFFFFFF00000000_{63..19}$||0||ci||si||$pa_{7..5}$||$0^4$
        WritePhysical(nca, 256, data)
    endif

    // retrieve data from cache
    if -bd then
        nca ← $0xFFFFFFFF00000000_{63..19}$||0||ci||si||$pa_{7..5}$||$0^4$
        data ← ReadPhysical(nca, 128)
    endif

    // write data into cache
    if (op=W) and bd and al then
        nca ← $0xFFFFFFFF00000000_{63..19}$||0||ci||si||$pa_{7..5}$||$0^4$
        data ← ReadPhysical(nca, 128)
        mdata ← $data_{127..8*(size+pa3..0)}$ || $wd_{8*(size+pa3..0)-1..8*pa3..0}$ || $data_{8*pa3..0..0}$
        WritePhysical(nca, 128, mdata)
    endif

    // prefetch into cache
    if al=bd and (cc=PF or cc=LS) then
        af ← 0 // abort fetch if af becomes 1
        for i ← 0 to 7
            if -$Tag[si]_i$ and -af then
                data,cen,xen ← AccessPhysical($pa_{63..8}$||$i_{2..0}$||0||$0^4$,256,cc,R,0)
                if cen then
                    nca ← $0xFFFFFFFF00000000_{63..19}$||0||ci||si||$i_{2..0}$||$0^4$
                    WritePhysical(nca, 256, data)
                    $Tag[si]_i$ ← 1
                    dt ← 1
                else
```

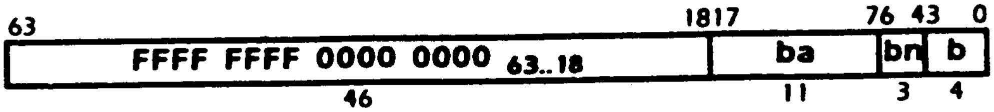
```
                    af ← 1
                endif
            endif
        endfor
    endif

    // cache tag writeback if dirty
    if dt then
        nt ← Tag[si || 1$^1$] || Tag[si || 0$^1$]
        WritePhysical(0xFFFFFFFF00000000$_{63..19}$ || CT || 0$^5$ || ci || si || 0$^4$, 128, nt)
    endif
enddef
```

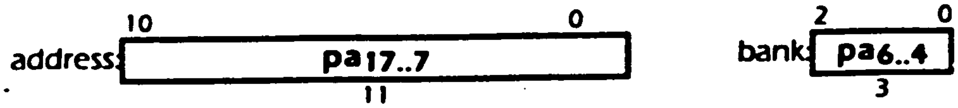
Physical address

The LOC data memory banks are accessed implicitly by cached memory accesses to any physical memory location as shown above. The LOC data memory banks are also accessed explicitly by uncached memory accesses to particular physical address ranges. The address mapping of these ranges is designed to facilitate use of a contiguous portion of the LOC cache as niche memory.

The physical address of a LOC hexlet for LOC address ba, bank bn, byte b is:



Within the explicit LOC data range, starting from a physical address pa17..0, the diagram below shows the LOC address (pa17..7) presented to LOC data bank (pa6..4).



The diagram below shows the LOC data memory bank and address referenced by byte address offsets in the explicit LOC data range. Note that this mapping includes the addresses use for LOC tags.

Byte offset	
0	bank 0, address 0
16	bank 1, address 0
32	bank 2, address 0
48	bank 3, address 0
64	bank 4, address 0
80	bank 5, address 0
96	bank 6, address 0
112	bank 7, address 0
128	bank 0, address 1
144	bank 1, address 1
160	bank 2, address 1
176	bank 3, address 1
192	bank 4, address 1
208	bank 5, address 1
224	bank 6, address 1
240	bank 7, address 1
...	...
262016	bank 0, address 2047
262032	bank 1, address 2047
262048	bank 2, address 2047
262064	bank 3, address 2047
262080	bank 4, address 2047
262096	bank 5, address 2047
262112	bank 6, address 2047
262128	bank 7, address 2047

Definition

```
def data ← AccessPhysicalLOC(pa,op,wd) as
    bank ← $pa_{6..4}$
    addr ← $pa_{17..7}$
    case op of
        R:
            rd ← LOCArray[bank][addr]
            crc ← LOCRedundancy[bank]
            data ← (crc and $rd_{130..2}$) or (~crc and $rd_{128..0}$)
            p[0] ← 0
            for i ← 0 to 128 by 1
                p[i+1] ← p[i] ^ $data_i$
            endfor
            if $ControlRegister_{61}$ and (p[129] ≠ 1) then
                raise CacheError
            endif
        W:
            p[0] ← 0
            for i ← 0 to 127 by 1
                p[i+1] ← p[i] ^ $wd_i$
            endfor
            $wd_{128}$ ← ~p[128]
            crc ← LOCRedundancy[bank]
            rdata ← ($crc_{126..0}$ and $wd_{126..0}$) or (~$crc_{126..0}$ and $wd_{128..2}$)
            LOCArray[bank][addr] ← $wd_{128..127}$ || rdata || $wd_{1..0}$
    endcase
enddef
```

Level One Cache Stress Control

LOC cells may be fabricated with marginal parameters, for which changes in clock timing or power supply voltage may cause these LOC cells to fail or pass. When testing the LOC while the part is in a normal circuit environment, rather than a special test environment with changeable power supply levels, cells with marginal parameters may not reliably fail testing.

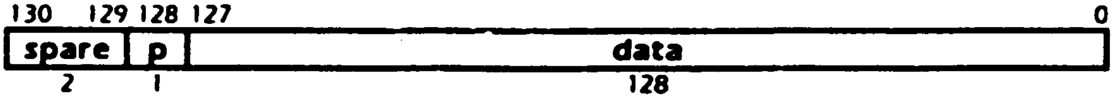
To combat this problem, two bits of the control register, **LOC stress**, may be set to stress the circuit environment while testing. Under normal operation, these bits are cleared (00), while during stress testing, one or more of these bits are set (01, 10, 11). Self-testing should be performed in each of the environment settings, and the detected failures combined together to produce a reliable test for cells with marginal parameters.

Level One Cache Redundancy

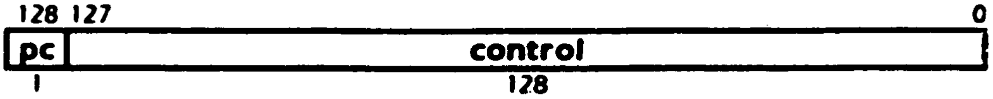
The LOC contains facilities that can be used to avoid minor defects in the LOC data array.

Each LOC bank has three additional bits of data storage for each 128 bits of memory data (for a total of 131 bits). One of these bits is used to retain odd parity over the 128 bits of memory data, and the other two bits are spare, which can be pressed into service by setting a non-zero value in the LOC redundancy control register for that bank.

Each row of a LOC bank contains 131 bits: 128 bits of memory data, one bit for parity, and two spare bits:



LOC redundancy control has 129 bits::



Each bit set in the control word causes the corresponding data bit to be selected from a bit address increased by two:

$$\textbf{output} \leftarrow (\textbf{data and} \sim\textbf{control}) \text{ or } ((\textbf{spare}_0 \mid\mid \textbf{p} \mid\mid \textbf{data}_{127..2}) \textbf{ and control})$$

$$\textbf{parity} \leftarrow (\textbf{p and} \sim\textbf{pc}) \text{ or } (\textbf{spare}_1 \textbf{ and pc})$$

The LOC redundancy control register has 129 bits, but is written with a 128-bit value. To set the pc bit in the LOC redundancy control, a value is written to the control with either bit

124 set (1) or bit 126 set (1). To set bit 124 of the LOC redundancy control, a value is written to the control with both bit 124 set (1) and 126 set (1). When the LOC redundancy control register is read, the process is reversed by selecting the pc bit instead of control bit 124 for the value of bit 124 if control bit 126 is zero (0).

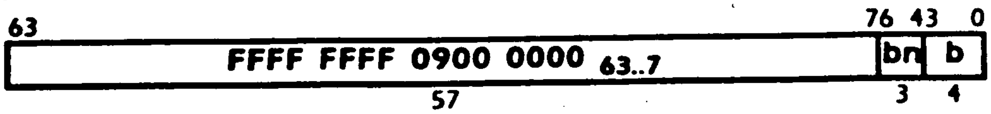
This system can remove one defective column at an even bit position and one defective column at an odd bit position within each LOC block. For each defective column location, x, LOC control bit must be set at bits x, x+2, x+4, x+6, ... If the defective column is in the parity location (bit 128), then set bit 124 only. The following table defines the control bits for parity, bit 126 and bit 124: (other control bits are same as values written)

$value_{126}$	$value_{124}$	pc	$control_{126}$	$control_{124}$
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
1	1	1	1	1

Physical address

The LOC redundancy controls are accessed explicitly by uncached memory accesses to particular physical address ranges.

The physical address of a LOC redundancy control for LOC bank bn, byte b is:



Definition

```
def data ← AccessPhysicalLOCRedundancy(pa,op,wd) as
    bank ← $pa_{6..4}$
    case op of
        R:
            rd ← LOCRedundancy[bank]
            data ← $rd_{127..125}$ || ($rd_{126}$ ? $rd_{124}$ : $rd_{128}$) || $rd_{123..0}$
        W:
            rd ← ($wd_{126}$ or $wd_{124}$) || $wd_{127..125}$ || ($wd_{126}$ and $wd_{124}$) || $wd_{123..0}$
            LOCRedundancy[bank] ← rd
    endcase
enddef
```

Memory Attributes

Fields in the LTB, GTB and cache tag control various attributes of the memory access in the specified region of memory. These include the control of cache consultation, updating, allocation, prefetching, coherence, ordering, victim selection, detail access, and cache prefetching.

Cache Control

The cache may be used in one of five ways, depending on a three-bit cache control field (cc) in the LTB and GTB. The cache control field may be set to one of seven states: NC, CD, WT, WA, PF, SS, and LS:

State		read		write		read/write	
		consult	allocate	update	allocate	victim	prefetch
No Cache	0	No	No	No	No	No	No
Cache Disable	1	Yes	No	Yes	No	No	No
Write Through	2	Yes	Yes	Yes	No	No	No
reserved	3						
Write Allocate	4	Yes	Yes	Yes	Yes	No	No
PreFetch	5	Yes	Yes	Yes	Yes	No	Yes
SubStream	6	Yes	Yes	Yes	Yes	Yes	No
LineStream	7	Yes	Yes	Yes	Yes	Yes	Yes

The Zeus processor controls cc as an attribute in the LTB and GTB, thus software may set this attribute for certain address ranges and clear it for others. A three-bit field indicates the choice of caching, according to the table above. The maximum of the three-bit cache control field (cc) values of the LTB and GTB indicates the choice of caching, according to the table above.

No Cache

No Cache (NC) is an attribute that can be set on a LTB or GTB translation region to indicate that the cache is to be not to be consulted. No changes to the cache state result from reads or writes with this attribute set, (except for accesses that directly address the cache via memory-mapped region).

Cache Disable

Cache Disable (CD) is an attribute that can be set on a LTB or GTB translation region to indicate that the cache is to be consulted and updated for cache lines which are already present, but no new cache lines or sub-blocks are to be allocated when the cache does not already contain the addressed memory contents.

The "Socket 7" bus also provides a mechanism for supporting chip sets to decide on each access whether data is to be cached, using the CACHE# and KEN# signals. Using these signals, external hardware may cause a region selected as **WT**, **WA** or **PF** to be treated as **CD**. This mechanism is only active on the first such access to a memory region if caching is enabled, as the cache may satisfy subsequent references without a bus transaction.

Write Through

Write Through (**WT**) is an attribute that can be set on a LTB or GTB translation region to indicate that the writes to the cache must also immediately update backing memory. Reads to addressed memory that is not present in the cache cause cache lines or sub-blocks to be allocated. Writes to addressed memory that is not present in the cache does not modify cache state.

The "Socket 7" bus also provides a mechanism for supporting chip sets to decide on each access whether data is to be written through, using the PWT and WB/WT# signals. Using these signals, external hardware may cause a region selected as **WA** or **PF** to be treated as **WT**. This mechanism is only active on the first write to each region of memory; as on subsequent references, if the cache line is in the Exclusive or Modified state and writeback caching is enabled on the first reference, no subsequent bus operation occurs, at least until the cache line is flushed.

Write Allocate

Write allocate (**WA**) is an attribute that can be set of a LTB or GTB translation region to indicate that the processor is to allocate a memory block to the cache when the data is not previously present in the cache and the operation to be performed is a store. Reads to addressed memory that is not present in the cache cause cache lines or sub-blocks to be allocated. For cacheable data, write allocate is generally the preferred policy, as allocating the data to the cache reduces further bus traffic for subsequent references (loads or stores) or the data. Write allocate never occurs for data which is not cached. A write allocate brings in the data immediately into the Modified state.

Other "socket 7" processors have the ability to inhibit write allocate to cached locations under certain conditions, related by the address range. K6, for example, can inhibit write allocate in the range of 15-16Mbyte, or for all addresses above a configurable limit with 4Mbyte granularity. Pentium has the ability to label address ranges over which write allocate can be inhibited.

PreFetch

Prefetch (**PF**) is an attribute that can be set on a LTB or GTB translation region to indicate that increased prefetching is appropriate for references in this region. Each program fetch, load or store to a cache line that or does not already contain all the sub-blocks causes a prefetch allocation of the remaining sub-blocks. Cache misses cause allocation of the requested sub-block and prefetch allocation of the remaining sub-blocks. Prefetching does not necessarily fill in the entire cache line, as prefetch memory references are performed at a lower priority to other cache and memory reference traffic. A limited number of prefetches (as low as one in the initial implementation) can be queued; the older prefetch requests are terminated as new ones are created.

In other respects, the **PF** attribute is handled in the manner of the **WA** attribute. Prefetching is considered an implementation-dependent feature, and an implementation may choose to implement region with the **PF** attribute exactly as with the **WA** attribute.

Implementations may perform even more aggressive prefetching in future versions. Data may be prefetched into the cache in regions that are cacheable, as a result of program fetches, loads or stores to nearby addresses. Prefetches may extend beyond the cache line associated with the nearby address. Prefetches shall not occur beyond the reach of the GTB entry associated with the nearby address. Prefetching is terminated if an attempted cache fill results in a bus response that is not cacheable. Prefetches are implementation-dependent behavior, and such behavior may vary as a result of other memory references or other bus activity.

SubStream

SubStream (SS) is an attribute that can be set on a LTB or GTB translation region to indicate that references in this region are to be selected as the next victim on a cache miss. In particular, cache misses, which normally place the cache line in the last-to-be-victim state, instead place the cache line in the first-to-be-victim state, except relative to cache lines in the I state.

In other respects, the SS attribute is handled in the manner of the WA attribute. SubStream is considered an implementation-dependent feature, and an implementation may choose to implement region with the SS attribute exactly as with the WA attribute.

The SubStream attribute is appropriate for regions which are large data structures in which the processor is likely to reference the memory data just once or a small number of times, but for which the cache permits the data to be fetched using burst transfers. By making it a priority for victimization, these references are less likely to interfere with caching of data for which the cache performs a longer-term storage function.

LineStream

LineStream (LS) is an attribute that can be set on a LTB or GTB translation region to indicate that references in this region are to be selected as the next victim on a cache miss, and to enable prefetching. In particular, cache misses, which normally place the cache line in the last-to-be-victim state, instead place the cache line in the first-to-be-victim state, except relative to cache lines in the I state.

In other respects, the LS attribute is handled in the manner of the PF attribute. LineStream is considered an implementation-dependent feature, and an implementation may choose to implement region with the SS attribute exactly as with the PF or WA attributes.

Like the SubStream attribute, the LineStream attribute is particularly appropriate for regions for whch large data structures are used in sequential fashion. By prefetching the entire cache line, memory traffic is performed as large sequential bursts of at least 256 bytes, maximizing the available bus utilization.

Cache Coherence

Cache coherency is maintained by using MESI protocols, for which each cache line (256 bytes) the cache data is kept in one of four states: **M, E, S, I**:

State		this Cache data	other Cache data	Memory data
Modified	3	Data is held exclusively in this cache.	No data is present in other caches.	The contents of main memory are now invalid.
Exclusive	2	Data is held exclusively in this cache.	No data is present in other caches.	Data is the same as the contents of main memory
Shared	1	Data is held in this cache, and possibly others.	Data is possibly in other caches.	Data is the same as the contents of main memory.
Invalid	0	No data for this location is present in the cache.	Data is possibly in other caches.	Data is possibly present in main memory.

The state is contained in the **mesi** field of the cache tag.

In addition, because the "Socket 7" bus performs block transfers and cache coherency actions on triclet (32 byte) blocks, each cache line also maintains 8 bits of triclet valid (tv) state. Each bit of tv corresponds to a triclet sub-block of the cache line; bit 0 for bytes 0..31, bit 1 for bytes 32..63, bit 2 for bytes 64..95, etc. If the tv bit is zero (0), the coherence state for that triclet is I, no matter what the value of the **mesi** field. If the tv bit is one (1), the coherence state is defined by the **mesi** field. If all the tv bits are cleared (0), the **mesi** field must also be cleared, indicating an invalid cache line.

Cache coherency activity generally follows the protocols defined by the "Socket 7" bus, as defined by Pentium and K6-2 documentation. However, because the coherence state of a cache line is represented in only 10 bits per 256 bytes (1.25 bits per triclet), a few state transistions are defined differently. The differences are a direct result of attempts to set triclets within a cache line to different MES states that cannot be represented. The data structure allows any triclet to be changed to the I state, so state transitions in this direction match the Pentium processor exactly.

On the Pentium processor, for a cache line in the M state, an external bus Inquiry cycle that does not require invalidation (INV=0) places the cache line in the S state. On the Zeus processor, if no other triclet in the cache line is valid, the **mesi** field is changed to S. If other triclets in the cache line are valid, the **mesi** field is left unchanged, and the tv bit for this triclet is turned off, effectively changing it to the I state.

On the Pentium processor, for a cache line in the E state, an external bus Inquiry cycle that does not require invalidation (INV=0) places the cache line in the S state. On the Zeus processor, the **mesi** field is changed to S. If other triclets in the cache line are valid, the MESI state is effectively changed to the S state for these other triclets.

On the Pentium processor, for a cache line in the S state, an internal store operation causes a write-through cycle and a transition to the E state. On the Zeus processor, the mesi field is changed to E. Other triclets in the cache line are invalidated by clearing the tv bits; the MESI state is effectively changed to the I state for these other triclets.

When allocating data into the cache due to a store operation, data is brought immediately into the Modified state, setting the mesi field to M. If the previous mesi field is S, other triclets which are valid are invalidated by clearing the tv bits. If the previous mesi field is E, other triclets are kept valid and therefore changed to the M state.

When allocating data into the cache due to a load operation, data is brought into the Shared state, if another processor reports that the data is present in its cache or the mesi field is already set to S, the Exclusive state, if no processor reports that the data is present in its cache and the mesi field is currently E or I, or the Modified state if the mesi field is already set to M. The determination is performed by driving PWT low and checking whether WB/WT# is sampled high; if so the line is brought into the Exclusive state. (See page 202 (184) of the K6-2 documentation).

Strong Ordering

Strong ordering (so) is an attribute which permits certain memory regions to be operated with strong ordering, in which all memory operations are performed exactly in the order specified by the program and others to be operated with weak ordering, in which some memory operations may be performed out of program order.

The Zeus processor controls strong ordering as an attribute in the LTB and GTB, thus software may set this attribute for certain address ranges and clear it for others. A one bit field indicates the choice of access ordering. A one (1) bit indicates strong ordering, while a zero (0) bit indicates weak ordering.

With weak ordering, the memory system may retain store operations in a store buffer indefinitely for later storage into the memory system, or until a synchronization operation to any address performed by the thread that issued the store operation forces the store to occur. Load operations may be performed in any order, subject to requirements that they be performed logically subsequent to prior store operations to the same address, and subsequent to prior synchronization operations to any address. Under weak ordering it is permitted to forward results from a retained store operation to a future load operation to the same address. Operations are considered to be to the same address when any bytes of the operation are in common. Weak ordering is usually appropriate for conventional memory regions, which are side-effect free.

With strong ordering, the memory system must perform load and store operations in the order specified. In particular, strong-ordered load operations are performed in the order specified, and all load operations (whether weak or strong) must be delayed until all previous strong-ordered store operations have been performed, which can have a significant performance impact. Strong ordering is often required for memory-mapped I/O regions, where store operations may have a side-effect on the value returned by loads to other addresses. Note that Zeus has memory-mapped I/O, such as the TB, for which the use of strong ordering is essential to proper operation of the virtual memory system.

The EWBE# signal in "Socket 7" is of importance in maintaining strong ordering. When a write is performed with the signal inactive, no further writes to E or M state lines may occur until the signal becomes active. Further details are given in Pentium documentation (K6-2 docmentation may not apply to this signal.)

Victim Selection

One bit of the cache tag, the **vs** bit, controls the selection of which set of the four sets at a cache address should next be chosen as a victim for cache line replacement. Victim selection (**vs**) is an attribute associated with LOC cache blocks. No **vs** bits are present in the LTB or GTB.

There are two hexlets of tag information for a cache line, and replacement of a set requires writing only one hexlet. To update priority information for victim selection by writing only one hexlet, information in each hexlet is combined by an exclusive-or. It is the nature of the exclusive-or function that altering either of the two hexlets can change the priority information.

Full victim selection ordering for four sets

There are 4•3•2•1 = 24 possible orderings of the four sets, which can be completely encoded in as few as 5 bits: 2 bits to indicate highest priority, 2 bits for second-highest priority, 1 bit for third-highest priority, and 0 bits for lowest priority. Dividing this up per set and duplicating per hexlet with the exclusive-or scheme above requires three bits per set, which suggests simply keeping track of the three-highest priority sets with 2 bits each, using 6 bits total and three bits per set.

Specifically, **vs** *bits from the four sets are combined to produce a 6-bit value:*

$$\mathbf{vsc} \leftarrow (\mathbf{vs}\,3 \;||\; \mathbf{vs}\,2)\,\hat{}\,(\mathbf{vs}\,1 \;||\; \mathbf{vs}\,0)$$

The highest priority for replacement is set $\mathbf{vsc}_{1..0}$*, second highest priority is set* $\mathbf{vsc}_{3..2}$*, third highest priority is set* $\mathbf{vsc}_{5..4}$*, and lowest priority is* $\mathbf{vsc}_{5..4}\,\hat{}\,\mathbf{vsc}_{3..2}\,\hat{}\,\mathbf{vsc}_{1..0}$*. When the highest priority set is replaced, it becomes the new lowest priority and the others are moved up, computing a new* **vsc** *by:*

$$\mathbf{vsc} \leftarrow \mathbf{vsc}_{5..4}\,\hat{}\,\mathbf{vsc}_{3..2}\,\hat{}\,\mathbf{vsc}_{1..0} \;||\; \mathbf{vsc}_{5..2}$$

When replacing set **vsc** *for a LineStream or SubStream replacement, the priority for replacement is unchanged unless another set contains the* **invalid** *MESI state, computing a new* **vsc** *by:*

$$\begin{aligned}\mathbf{vsc} \leftarrow\; & \mathbf{mesi}\;\mathbf{vsc}_{5..4}\,\hat{}\,\mathbf{vsc}_{3..2}\,\hat{}\,\mathbf{vsc}_{1..0} = I)\; ?\; \mathbf{vsc}_{5..4}\,\hat{}\,\mathbf{vsc}_{3..2}\,\hat{}\,\mathbf{vsc}_{1..0} \;||\; \mathbf{vsc}_{5..2} :\\ & (\mathbf{mesi}\;\mathbf{vsc}_{5..4} = I)\; ?\; \mathbf{vsc}_{1..0} \;||\; \mathbf{vsc}_{5..2} :\\ & (\mathbf{mesi}\;\mathbf{vsc}_{3..2} = I)\; ?\; \mathbf{vsc}_{5..4} \;||\; \mathbf{vsc}_{1..0} \;||\; \mathbf{vsc}_{3..2} :\\ & \mathbf{vsc}\end{aligned}$$

Cache flushing and invalidations can cause cache lines to be cleared out of sequential order. Flushing or invalidating a cache line moves that set to highest priority. If that set is already highest priority, the vsc is unchanged. If the set was second or third highest or lowest priority, the vsc is changed to move that set to highest priority, moving the others down.

$$\mathbf{vsc} \leftarrow ((\mathbf{fs}=\mathbf{vsc}_{1..0} \text{ or } \mathbf{fs}=\mathbf{vsc}_{3..2})\ ?\ \mathbf{vsc}_{5..4} : \mathbf{vsc}_{3..2}) \mid\mid (\mathbf{fs}=\mathbf{vsc}_{1..0}\ ?\ \mathbf{vsc}_{3..2} : \mathbf{vsc}_{1..0}) \mid\mid \mathbf{fs}$$

When updating the hexlet containing **vs[1]** *and* **vs[0]**, *the new values of* **vs[1]** *and* **vs[0]** *are:*

$$\mathbf{vs}[1] \leftarrow \mathbf{vs}[3] \wedge \mathbf{vsc}_{5..3}$$

$$\mathbf{vs}[0] \leftarrow \mathbf{vs}[2] \wedge \mathbf{vsc}_{2..0}$$

When updating the hexlet containing **vs[3]** *and* **vs[2]**, *the new values of* **vs[3]** *and* **vs[2]** *are:*

$$\mathbf{vs}[3] \leftarrow \mathbf{vs}[1] \wedge \mathbf{vsc}_{5..3}$$

$$\mathbf{vs}[2] \leftarrow \mathbf{vs}[0] \wedge \mathbf{vsc}_{2..0}$$

Software must initialize the **vs** *bits to a legal, consistent state. For example, to set the priority (highest to lowest) to (0, 1, 2, 3),* **vsc** *must be set to 0b100100. There are many legal solutions that yield this* **vsc** *value, such as* **vs[3]** ← *0,* **vs[2]** ← *0,* **vs[1]** ← *4,* **vs[0]** ← *4.*

Simplified victim selection ordering for four sets

However, the orderings are simplified in the first Zeus implementation, to reduce the number of **vs** bits to one per set, keeping a two bit vsc state value:

$$\mathbf{vsc} \leftarrow (\mathbf{vs}[3] \mid\mid \mathbf{vs}[2]) \wedge (\mathbf{vs}[1] \mid\mid \mathbf{vs}[0])$$

The highest priority for replacement is set **vsc**, second highest priority is set **vsc+1**, third highest priority is set **vsc+2**, and lowest priority is **vsc+3**. When the highest priority set is replaced, it becomes the new lowest priority and the others are moved up. Priority is given to sets with **invalid** MESI state, computing a new **vsc** by:

$$\begin{aligned}\mathbf{vsc} \leftarrow \mathbf{mesi}[\mathbf{vsc}+1]=\mathbf{I})\ ?\ \mathbf{vsc}+1 &:\\ (\mathbf{mesi}[\mathbf{vsc}+2]=\mathbf{I})\ ?\ \mathbf{vsc}+2 &:\\ (\mathbf{mesi}[\mathbf{vsc}+3]=\mathbf{I})\ ?\ \mathbf{vsc}+3 &:\\ \mathbf{vsc}+1 &\end{aligned}$$

When replacing set **vsc** for a LineStream or SubStream replacement, the priority for replacement is unchanged, unless another set contains the **invalid** MESI state, computing a new **vsc** by:

$$\begin{aligned}\mathbf{vsc} \leftarrow \mathbf{mesi}[\mathbf{vsc}+1]=\mathbf{I})\ ?\ \mathbf{vsc}+1 &:\\ (\mathbf{mesi}[\mathbf{vsc}+2]=\mathbf{I})\ ?\ \mathbf{vsc}+2 &:\\ (\mathbf{mesi}[\mathbf{vsc}+3]=\mathbf{I})\ ?\ \mathbf{vsc}+3 &:\\ \mathbf{vsc} &\end{aligned}$$

Cache flushing and invalidations can cause cache sets to be cleared out of sequential order. If the current highest priority for replacement is a valid set, the flushed or invalidated set is made highest priority for replacement.

$$\mathbf{vsc} \leftarrow (\mathbf{mesi}[\mathbf{vsc}]=\mathbf{I})\ ?\ \mathbf{vsc} : \mathbf{fs}$$

When updating the hexlet containing **vs**[1] and **vs**[0], the new values of **vs**[1] and **vs**[0] are:

$$\mathbf{vs}[1] \leftarrow \mathbf{vs}[3]\ \hat{}\ \mathbf{vsc}_1$$

$$\mathbf{vs}[0] \leftarrow \mathbf{vs}[2]\ \hat{}\ \mathbf{vsc}_0$$

When updating the hexlet containing **vs**[3] and **vs**[2], the new values of **vs**[3] and **vs**[2] are:

$$\mathbf{vs}[3] \leftarrow \mathbf{vs}[1]\ \hat{}\ \mathbf{vsc}_1$$

$$\mathbf{vs}[2] \leftarrow \mathbf{vs}[0]\ \hat{}\ \mathbf{vsc}_0$$

Software must initialize the **vs** bits, but any state is legal. For example, to set the priority (highest to lowest) to (0, 1, 2, 3), **vsc** must be set to 0b00. There are many legal solutions that yield this **vsc** value, such as **vs**[3] ← 0, **vs**[2] ← 0, **vs**[1] ← 0,**vs**[0] ← 0.

Full victim selection ordering for additional sets

*To extend the full-victim-ordering scheme to eight sets, 3*7=21 bits are needed, which divided among two tags is 11 bits per tag. This is somewhat generous, as the minimum required is 8*7*6*5*4*3*2*1=40320 orderings, which can be represented in as few as 16 bits. Extending the full-victim-ordering four-set scheme above to represent the first 4 priorities in binary, but to use 2 bits for each of the next 3 priorities requires 3+3+3+3+2+2+2 = 18 bits. Representing fewer distinct orderings can further reduce the number of bits used. As an extreme example, using the simplified scheme above with eight sets requires only 3 bits, which divided among two tags is 2 bits per tag.*

Victim selection without LOC tag bits

At extreme values of the **niche limit** register (**nl** in the range 121..124), the bit normally used to hold the **vs** bit is usurped for use as a physical address bit. Under these conditions, no **vsc** value is maintained per cache line, instead a single, global **vsc** value is used to select victims for cache replacement. In this case, the cache consists of four lines, each with four sets. On each replacement a new **si** valus is computed from:

$$\mathbf{gvsc} \leftarrow \mathbf{gvsc} + 1$$

$$\mathbf{si} \leftarrow \mathbf{gvsc}\ \hat{}\ \mathbf{pa}_{11..10}$$

The algorithm above is designed to utilize all four sets on sequential access to memory.

Victim selection encoding LOC tag bits

At even more extreme values of the niche limit register (nl in the range 125..127), not only is the bit normally used to hold the vs bit is usurped for use as a physical address bit, but there is a deficit of one or two physical address bits. In this case, the number of sets can be reduced to encode physical address bits into the victim selection, allowing the choice of set to indicate physical address bits 9 or bits 9..8. On each replacement a new vsc valus is computed from:

$$\mathbf{gvsc} \leftarrow \mathbf{gvsc} + 1$$

$$\mathbf{si} \leftarrow \mathbf{pa}_9 \;||\; (\mathbf{nl}=127)\; ?\; \mathbf{pa}_8 : \mathbf{gvsc}\hat{\ }\mathbf{pa}_{10}$$

The algorithm above is designed to utilize all four sets on sequential access to memory.

Detail Access

Detail access is an attribute which can be set on a cache block or translation region to indicate that software needs to be consulted on each potential access, to determine whether the access should proceed or not. Setting this attribute causes an exception trap to occur, by which software can examine the virtual address, by for example, locating data in a table, and if indicated, causes the processor to continue execution. In continuing, ephemeral state is set upon returning to the re-execution of the instruction that prevents the exception trap from recurring on this particular re-execution only. The ephemeral state is cleared as soon as the instruction is either completed or subject to another exception, so DetailAccess exceptions can recur on a subsequent execution of the same instruction. Alternatively, if the access is not to proceed, execution has been trapped to software at this point, which can abort the thread or take other corrective action.

The detail access attribute permits specification of access parameters over memory region on arbitrary byte boundaries. This is important for emulators, which must prevent store access to code which has been translated, and for simulating machines which have byte granularity on segment boundaries. The detail access attribute can also be applied to debuggers, which have the need to set breakpoints on byte level data, or which may use the feature to set code breakpoints on instruction boundaries without altering the program code, enabling breakpoints on code contained in ROM.

A one bit field indicates the choice of detail access. A one (1) bit indicates detail access, while a zero (0) bit indicates no detail access. Detail access is an attribute that can be set by the LTB, the GTB, or a cache tag.

The table below indicates the proper status for all potential values of the detail access bits in the LTB, GTB, and Tag:

LTB	GTB	Tag	status
0	0	0	OK - normal
0	0	1	AccessDetailRequiredByTag
0	1	0	AccessDetailRequiredByGTB
0	1	1	OK - GTB inhibited by Tag
1	0	0	AccessDetailRequiredByLTB
1	0	1	OK - LTB inhibited by Tag
1	1	0	OK - LTB inhibited by GTB
1	1	1	AccessDetailRequiredByTag
0	Miss		GTBMiss
1	Miss		AccessDetailRequiredByLTB
0	0	Miss	Cache Miss
0	1	Miss	AccessDetailRequiredByGTB
1	0	Miss	AccessDetailRequiredByLTB
1	1	Miss	Cache Miss

The first eight rows show appropriate activities when all three bits are available. The detail access attributes for the LTB, GTB, and cache tag work together to define whether and which kind of detail access exception trap occurs. Generally, setting a single attribute bit causes an exception, while setting two bits inhibits such exceptions. In this way, a detail access exception can be narrowed down to cause an exception over a specified region of memory: Software generally will set the cache tag detail access bit only for regions in which the LTB or GTB also has a detail access bit set. Because cache activity may flush and refill cache lines implicity, it is not generally useful to set the cache tag detail access bit alone, but if this occurs, the AccessDetailRequiredByTag exception catches such an attempt.

The next two rows show appopropriate activities on a GTB miss. On a GTB miss, the detail access bit in the GTB is not present. If the LTB indicates detail access and the GTB misses, the AccessDetailRequiredByLTB exception should be indicated. If software continues from the AccessDetailRequiredByLTB exception and has not filled in the GTB, the GTBMiss exception happens next. Since the GTBMiss exection is not a continuation exception, a re-execution after the GTBMiss exception can cause a reoccurence of the AccessDetailRequiredByLTB exception. Alternatively, if software continues from the AccessDetailRequiredByLTB exception and has filled in the GTB, the AccessDetailRequiredByLTB exception is inhibited for that reference, no matter what the status of the GTB and Tag detail bits, but the re-executed instruction is still subject to the AccessDetailRequiredByGTB and AccessDetailRequiredByTag exceptions.

The last four rows show appropriate activities for a cache miss. On a cache miss, the detail access bit in the tag is not present. If the LTB or GTB indicates detail access and the cache misses, the AccessDetailRequiredByLTB or AccessDetailRequiredByGTB exception should be indicated. If software continues from these exceptions and has not filled in the cache, a cache miss happens next. If software continues from the AccessDetailRequiredByLTB or AccessDetailRequiredByGTB exception and has filled in the cache, the previous exception is inhibited for that reference, no matter what the status of the Tag detail bit, but is still subject to the AccessDetailRequiredByTag exception. When the detail bit must be created from a cache miss, the intial value filled in is zero. Software may set the bit, thus turning off AccessDetailRequired exceptions per cache line. If the cache line is flushed and refilled, the detail access bit in the cache tag is again reset to zero, and another AccessDetailRequired exception occurs.

Settings of the niche limit parameter to values that require use of the da bit in the LOC tag for retaining the physical address usurp the capability to set the Tag detail access bit. Under such conditions, the Tag detail access bit is effectively always zero (0), so it cannot inhibit AccessDetailRequiredByLTB, inhibit AccessDetailRequiredByGTB, or cause AccessDetailRequiredByTag.

The execution of a Zeus instruction has a reference to one quadlet of instruction, which may be subject to the DetailAccess exceptions, and a reference to data, which may be unaligned or wide. These unaligned or wide references may cross GTB or cache boundaries, and thus involve multiple separate reference that are combined together, each of which may be subject to the DetailAccess exception. There is sufficient information in the DetailAccess exception handler to process unaligned or wide references.

The implementation is free to indicate DetailAccess exceptions for unaligned and wide data references either in combined form, or with each sub-reference separated. For example, in an unaligned reference that crosses a GTB or cache boundary, a DetailAccess exception may be indicated for a portion of the reference. The exception may report the virtual address and size of the complete reference, and upon continuing, may inhibit reoccurrence of the DetailAccess exception for any portion of the reference. Alternatively, it may report the virtual address and size of only a reference portion and inhibit reoccurrence of the DetailAccess exception for only that portion of the reference, subject to another DetailAccess exception occurring for the remaining portion of the reference.

Micro Translation Buffer

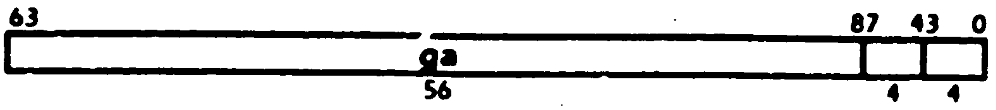
The Micro Translation Buffer (MTB) is an implementation-dependent structure which reduces the access traffic to the GTB and the LOC tags. The MTB contains and caches information read from the GTB and LOC tags, and is consulted on each access to the LOC.

To access the LOC, a global address is supplied to the Micro-Translation Buffer (MTB), which associatively looks up the global address into a table holding a subset of the LOC tags. In addition, each table entry contains the physical address bits 14..8 (7 bits) and set identifier (2 bits) required to access the LOC data.

In the first Zeus implementation, there are two MTB blocks - MTB 0 is used for threads 0 and 1, and MTB 1 is used for threads 2 and 3. Per clock cycle, each MTB block can check for 4 simultaneous references to the LOC. Each MTB block has 16 entries.

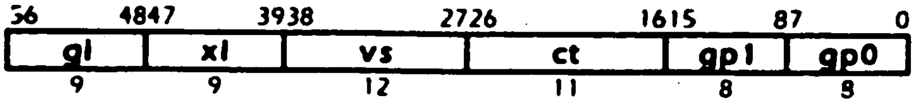
Each MTB entry consists of a bit less than 128 bits of information, including a 56-bit global address tag, 8 bits of privilege level required for read, write, execute, and gateway access, a detail bit, and 10 bits of cache state indicating for each triclet (32 bytes) sub-block, the MESI state.

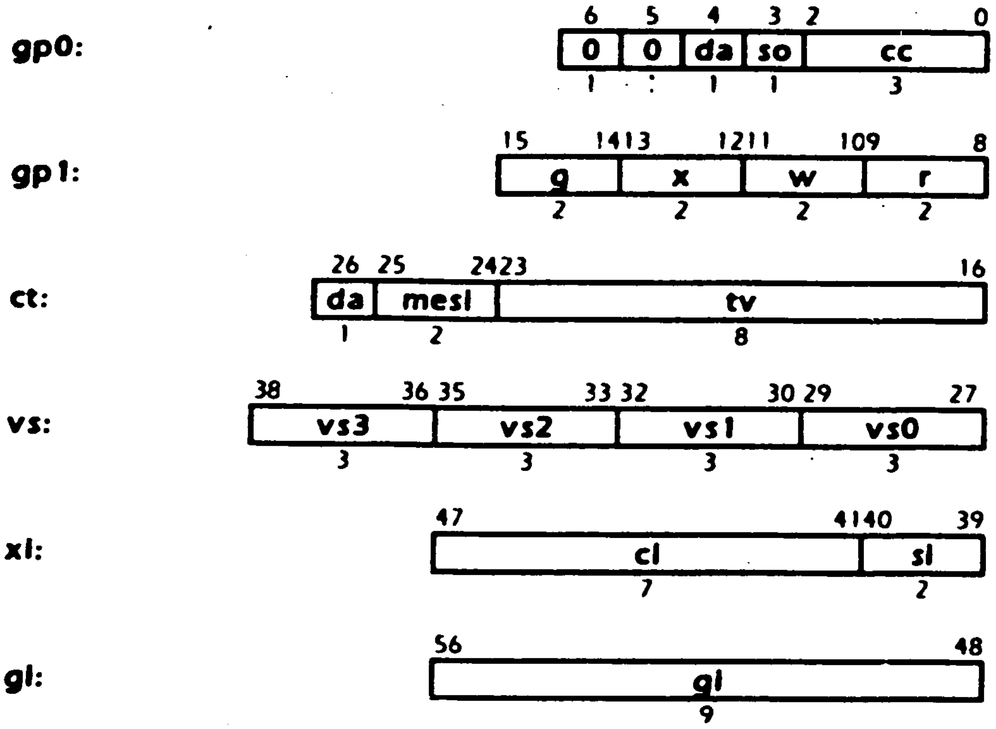
Match



Output

The output of the MTB combines physical address and protection information from the GTB and the referenced cache line.





The meaning of the fields are given by the following table:

name	size	meaning
ga	56	global address
gi	9	GTB index
ci	7	cache index
si	2	set index
vs	12	victim select
da	1	detail access (from cache line)
mesi	2	coherency: modified (3), exclusive (2), shared (1), invalid (0)
tv	8	triclet valid (1) or invalid (0)
g	2	minimum privilege required for gateway access
x	2	minimum privilege required for execute access
w	2	minimum privilege required for write access
r	2	minimum privilege required for read access
0	1	reserved
da	1	detail access (from GTB)
so	1	strong ordering
cc	3	cache control

With an MTB hit, the resulting cache index (14..8 from the MTB, bit 7 from the LA) and set identifier (2 bits from the MTB) are applied to the LOC data bank selected from bits 6..4 of the GVA. The access protection information (pr and rwxg) is supplied from the MTB.

With an MTB (and BTB) miss, a victim entry is selected for replacement. The MTB and BTB are always clean, so the victim entry is discarded without a writeback. The GTB (described below) is referenced to obtain a physical address and protection information. Depending on the access information in the GTB, either the MTB or BTB is filled.

Note that the processing of the physical address $pa_{14..8}$ against the niche limit **nl** can be performed on the physical address from the GTB, producing the LOC address, **ci**. The LOC address, after processing against the **nl** is placed into the MTB directly, reducing the latency of an MTB hit.

Four tags are fetched from the LOC tags and compared against the PA to determine which of the four sets contain the data. If one of the four sets contains the correct physical address, a victim MTB entry is selected for replacement, the MTB is filled and the LOC access proceeds. If none of the four sets is a hit, an LOC miss occurs.

MTB miss GTB cam LOC tag MTB fill

MTB victim

LOC miss

The operation of the MTB is largely not visible to software - hardware mechanisms are responsible for automatically initializing, filling and flushing the MTB. Activity that modifies the GTB or LOC tag state may require that one or more MTB entries are flushed.

A write to the GTBUpdate register that updates a matching entry, a write to the GTBUpdateFill register, or a direct write to the GTB all flush relevant entries from the MTB. MTB flushing is accomplished by searching MTB entries for values that match on the gi field with the GTB entry that has been modified. Each such matching MTB entry is flushed.

The MTB is kept synchronous with the LOC tags, particularly with respect to MESI state. On an LOC miss or LOC snoop, any changes in MESI state update (or flush) MTB entries which physically match the address. If the MTB may contain less than the full physical address: it is sufficient to retain the LOC physical address (ci || v || si).

Block Translation Buffer

Zeus has a per thread "Block Translation Buffer" (BTB). The BTB retains GTB information for uncached address blocks. The BTB is used in parallel with the MTB - exactly one of the BTB or MTB may translate a particular reference. When both the BTB and MTB miss, the GTB is consulted, and depending on the result, the block is filled into either the MTB or BTB as appropriate. In the first Zeus implementation, the BTB has 2 entries for each thread.

BTB entries cover any power-of-two granularity, as they retain the size information from the GTB. BTB entries contain no MESI state, as they only contain uncached blocks.

Each BTB entry consists of 128 bits of information, containing the same information in the same format as a GTB entry.

Niche blocks are indicated by GTB information, and correspond to blocks of data that are retained in the LOC and never miss. A special physical address range indicates niche blocks. For this address range, the BTB enables use of the LOC as a niche memory, generating the "set select" address bits from low-order address bits. There is no checking of the LOC tags for consistent use of the LOC as a niche - the nl field must be preset by software so that LOC cache replacement never claims the LOC niche space, and only BTB miss and protection bits prevent software from using the cache portion of the LOC as niche.

Other address ranges include other on-chip resources, such as bus interface registers, the control register and status register, as well as off-chip memory, accessed through the bus interface. Each of these regions are accessible as uncached memory.

Program Translation Buffer

Later implementations of Zeus may optionally have a per thread "Program Translation Buffer" (PTB). The PTB retains GTB and LOC cache tag information. The PTB enables generation of LOC instruction fetching in parallel with load/store fetching. The PTB is updated when instruction fetching crosses a cache line boundary (each 64 instructions in straight-line code). The PTB functions similarly to a one-entry MTB, but can use the sequential nature of program code fetching to avoid checking the 56-bit match. The PTB is flushed at the same time as the MTB.

The initial implementation of Zeus has no PTB - the MTB suffices for this function.

Global Virtual Cache

The initial implementation of Zeus contains cache which is both indexed and tagged by a physical address. Other prototype implementations have used a global vitual address to index and/or tag an internal cache. This section will define the required characteristics of a global vitually-indexed cache. TODO

Memory Interface

Dedicated hardware mechanisms are provided to fetch data blocks in the levels zero and one caches, provided that a matching entry can be found in the MTB or GTB (or if the MMU is disabled). Dedicated hardware mechanisms are provided to store back data blocks in the level zero and one caches, regardless of the state of the MTB and GTB. When no entry is to be found in the GTB, an exception handler is invoked either to generate the required information from the virtual address, or to place an entry in the GTB to provide for automatic handling of this and other similarly addressed data blocks.

The initial implementation of Zeus accesses the remainder of the memory system through the "Socket 7" interface. Via this interface, Zeus accesses a secondary cache, DRAM memory, external ROM memory, and an I/O system The size and presence of the secondary cache and the DRAM memory array, and the contents of the external ROM memory and the I/O system are variables in the processor environment.

Microarchitecture

Each thread has two address generation units, capable of producing two aligned, or one unaligned load or store operation per cycle. Alternatively, these units may produce a single load or store address and a branch target address.

Each thread has a LTB, which translates the two addresses into global virtual addresses.

Each pair of threads has a MTB, which looks up the four references into the LOC. The PTB provides for additional references that are program code fetches.

In parallel with the MTB, these four references are combined with the four references from the other thread pair and partitioned into even and odd hexlet references. Up to four references are selected for each of the even and odd portions of the LZC. One reference for each of the eight banks of the LOC (four are even hexlets; four are odd hexlets) are selected from the eight load/store/branch references and the PTB references.

Some references may be directed to both the LZC and LOC, in which case the LZC hit causes the LOC data to be ignored. An LZC miss which hits in the MTB is filled from the LOC to the LZC. An LZC miss which misses in the MTB causes a GTB access and LOC tag access, then an MTB fill and LOC access, then an LZC fill.

Priority of access: (highest/lowest) cache dump, cache fill, load, program, store.

Snoop

The "Socket 7" bus requires certain bus accesses to be checked against on-chip caches. On a bus read, the address is checked against the on-chip caches, with accesses aborted when requested data is in an internal cache in the M state, and the E state, the internal cache is changed to the S state. On a bus write, data written must update data in on-chip caches. To meet these requirements, physical bus addresses must be checked against the LOC tags.

The S7 bus requires that responses to inquire cycles occur with fixed timing. At least with certain combinations of bus and processor clock rate, inquire cycles will require top priority to meet the inquire response timing requirement.

Synchronization operations must take into account bus activity - generally a synchronization operation can only proceed on cached data which is in Exclusive or Modified – if cached data in Shared state, ownership must be obtained. Data that is not cached must be accessed using locked bus cycles.

Load

Load operations require partitioning into reads that do not cross a hexlet (128 bit) boundary, checking for store conflicts, checking the LZC, checking the LOC, and reading from memory. Execute and Gateway accesses are always aligned and since they are smaller than a hexlet, do not cross a hexlet boundary.

Note: S7 processors perform unaligned operations LSB first, MSB last, up to 64 bits at a time. Unaligned 128 bit loads need 3 64-bit operations, LSB, octlet, MSB. Transfers which are smaller than a hexlet but larger than an octlet are further divided in the S7 bus unit.

Definition

```
def data ← LoadMemoryX(ba,la,size,order)
     assert (order = L) and ((la and (size/8-1)) = 0) and (size = 32)
     hdata ← TranslateAndCacheAccess(ba,la,size,X,0)
     data ← hdata$_{31+8*(la and 15)..8*(la and 15)}$
enddef

def data ← LoadMemoryG(ba,la,size,order)
     assert (order = L) and ((la and (size/8-1)) = 0) and (size = 64)
     hdata ← TranslateAndCacheAccess(ba,la,size,G,0)
     data ← hdata$_{63+8*(la and 15)..8*(la and 15)}$
enddef

def data ← LoadMemory(ba,la,size,order)
     if (size > 128) then
          data0 ← LoadMemory(ba, la,size/2, order)
          data1 ← LoadMemory(ba, la+(size/2), size/2, order)
          case order of
               L:
                    data ← data1 || data0
               B:
```

```
            data ← data0 || data1
        endcase
    else
        bs ← 8*$la_{4..0}$
        be ← bs + size
        if be > 128 then
            data0 ← LoadMemory(ba, la, 128 – bs, order)
            data1 ← LoadMemory(ba, ($la_{63..5}$ + 1) || $0^4$, be - 128, order)
            case order of
                L:
                    data ← (data1 || data0)
                B:
                    data ← (data0 || data1)
            endcase
        else
            hdata ← TranslateAndCacheAccess(ba,la,size,R,0)
            for i ← 0 to size-8 by 8
                j ← bs + ((order=L) ? i : size-8-i)
                $data_{i+7..i}$ ← $hdata_{j+7..j}$
            endfor
        endif
    endif
enddef
```

Store

Store operations requires partitioning into stores less than 128 bits that do not cross hexlet boundaries, checking for store conflicts, checking the LZC, checking the LOC, and storing into memory.

Definition

```
def StoreMemory(ba,la,size,order,data)
    bs ← 8*$la_{4..0}$
    be ← bs + size
    if be > 128 then
        case order of
            L:
                data0 ← $data_{127-bs..0}$
                data1 ← $data_{size-1..128-bs}$
            B:
                data0 ← $data_{size-1..be-128}$
                data1 ← $data_{be-129..0}$
        endcase
        StoreMemory(ba, la, 128 – bs, order, data0)
        StoreMemory(ba, ($la_{63..5}$ + 1) || $0^4$, be - 128, order, data1)
    else
        for i ← 0 to size-8 by 8
            j ← bs + ((order=L) ? i : size-8-i)
            $hdata_{j+7..j}$ ← $data_{i+7..i}$
        endfor
        xdata ← TranslateAndCacheAccess(ba, la, size, W, hdata)
        endif
enddef
```

Memory

Memory operations require first translating via the LTB and GTB, checking for access exceptions, then accessing the cache.

Definition

```
def hdata ← TranslateAndCacheAccess(ba,la,size,rwxg,hwdata)
    if $ControlRegister_{62}$ then
        case rwxg of
            R:
                at ← 0
            W:
                at ← 1
            X:
                at ← 2
            G:
                at ← 3
        endcase
        rw ← (rwxg=W) ? W : R
        ga,LocalProtect ← LocalTranslation(th,ba,la,pl)
        if $LocalProtect_{9+2*at..8+2*at}$ < pl then
            raise AccessDisallowedByLTB
        endif
        lda ← $LocalProtect_4$
        pa,GlobalProtect ← GlobalTranslation(th,ga,pl,lda)
        if $GlobalProtect_{9+2*at..8+2*at}$ < pl then
            raise AccessDisallowedByGTB
        endif
        cc ← ($LocalProtect_{2..0}$ > $GlobalProtect_{2..0}$) ? $LocalProtect_{2..0}$ : $GlobalProtect_{2..0}$
        so ← $LocalProtect_3$ or $GlobalProtect_3$
        gda ← $GlobalProtect_4$
        hdata,TagProtect ← LevelOneCacheAccess(pa,size,lda,gda,cc,rw,hwdata)
        if (lda ^ gda ^ TagProtect) = 1 then
            if TagProtect then
                PerformAccessDetail(AccessDetailRequiredByTag)
            elseif gda then
                PerformAccessDetail(AccessDetailRequiredByGlobalTB)
            else
                PerformAccessDetail(AccessDetailRequiredByLocalTB)
            endif
        endif
    else
        case rwxg of
            R, X, G:
                hdata ← ReadPhysical(la,size)
            W:
                WritePhysical(la,size,hwdata)
        endcase
    endif
enddef
```

Bus interface

The initial implementation of the Zeus processor uses a "Super Socket 7 compatible" (SS7) bus interface, which is generally similar to and compatible with other "Socket 7" and "Super Socket 7" processors such as the Intel Pentium, Pentium with MMX Technology; AMD K6, K6-II, K6-III; IDT Winchip C6, 2, 2A, 3, 4; Cyrix 6x86, etc. and other "Socket 7" chipsets listed below.

The SS7 bus interface behavior is quite complex, but well-known due to the leading position of the Intel Pentium design. This document does not yet contain all the detailed information related to this bus, and will concentrate on the differences between the Zeus SS7 bus and other designs. For functional specification and pin interface behavior, the *Pentium Processor Family Developer's Manual*[27] is a primary reference. For 100 MHz SS7 bus timing data, the *AMD K6-2 Processor Data Sheet*[28] is a primary reference.

Motherboard Chipsets

The following motherboard chipsets are designed for the 100 MHz "Socket 7" bus:

Manufacturer	Website	Chipset	clock rate	North bridge	South bridge
VIA technologies, Inc.	www.via.com.tw	Apollo MVP3	100 MHz	vt82c598at[29]	vt82c598b
Silicon Integrated Systems	www.sis.com.tw	SiS 5591/5592	75 MHz	SiS 5591[30]	SiS 5595
Acer Laboratories, Inc.	www.acerlabs.com	Ali Aladdin V	100 MHz	M1541[31]	M1543C

The following processors are designed for a "Socket 7" bus:

Manufacturer	Website	Chips	clock rate
Advanced Micro Devices	www.amd.com	K6-2	100 MHz
Advanced Micro Devices	www.amd.com	K6-3	100 MHz
Intel	www.intel.com	Pentium MMX	66 MHz
IDT/Centaur	www.winchip.com	Winchip C6	75 MHz
IDT/Centaur	www.winchip.com	Winchip 2	100 MHz
IDT/Centaur	www.winchip.com	Winchip 2A	100 MHz
IDT/Centaur	www.winchip.com	Winchip 4	100 MHz
NSM/Cyrix	www.cyrix.com		

[27] http://home.microunity.com/~craig/standards/intel/intel-pentium-family-developers-manual-1997-24142805.pdf

[28] http://home.microunity.com/~craig/standards/amd/amd-k62-data-sheet-21850c.pdf

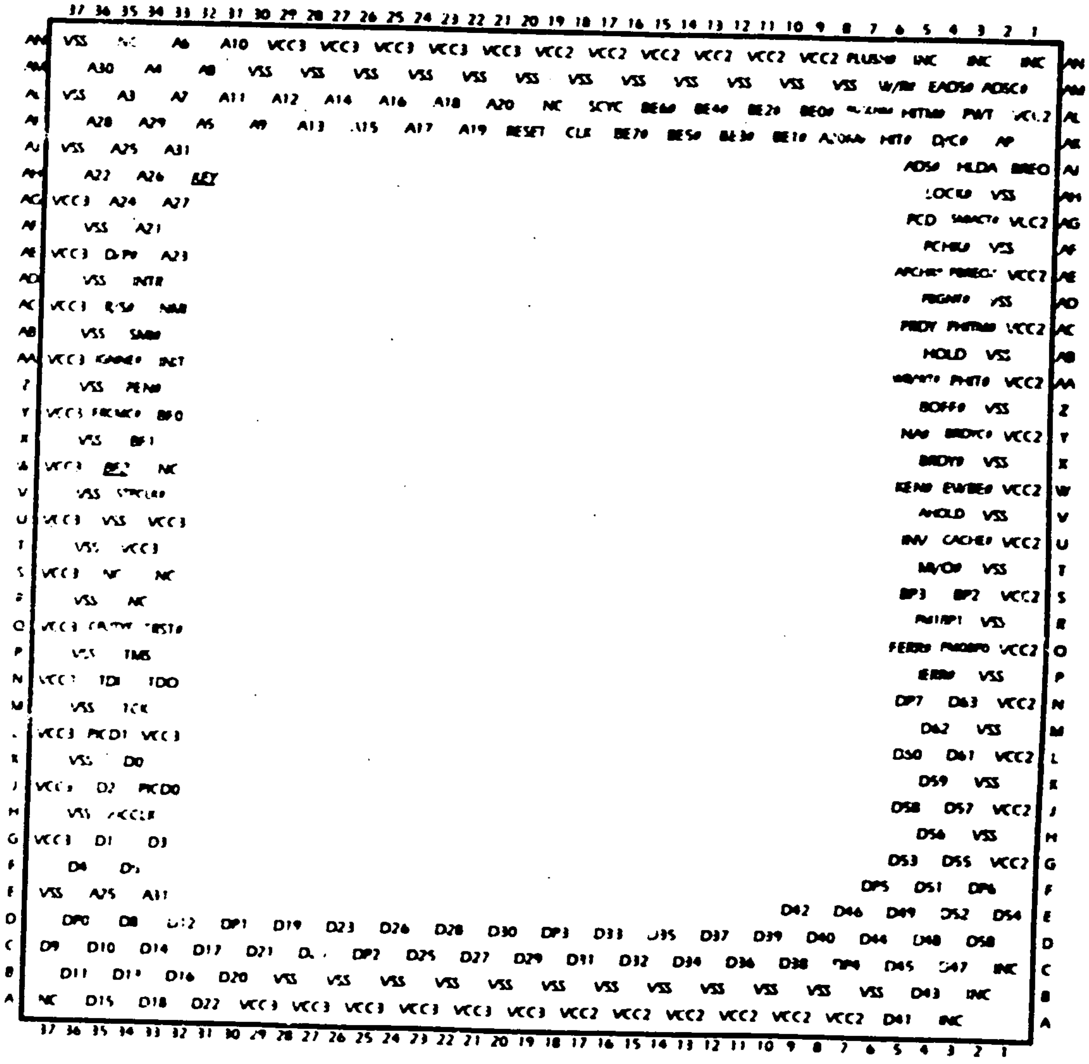
[29] http://home.microunity.com/~craig/standards/via/via-apollo-mvp3-vt82c598at-598.pdf

[30] http://home.microunity.com/~craig/standards/sis/5591ds10.doc

[31] http://home.microunity.com/~craig/standards/acer/alad4m5pb.htm

Pinout

In the diagram below, signals which are different from Pentium pinout, are indicated by italics and underlining. Generally, other Pentium-compatible processors (such as the AMD K6-2) define these signals.


37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
AN VSS NC A6 A10 VCC3 VCC3 VCC3 VCC3 VCC3 VCC2 VCC2 VCC2 VCC2 VCC2 VCC2 FLUSH# INC INC INC
AM A30 A4 A8 VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS W/R# EADS# ADSC# AN
AL VSS A3 A7 A11 A12 A14 A16 A18 A20 NC SCYC BE6# BE4# BE2# BE0# HITM# PWT VCC2 AM
AK A28 A29 A5 A9 A13 A15 A17 A19 RESET CLK BE7# BE5# BE3# BE1# A20M# HIT# D/C# AP AL
AJ VSS A25 A31 AK
AH A22 A26 KEY ADS# HLDA BREQ AJ
AG VCC3 A24 A27 LOCK# VSS AH
AF VSS A21 PCD SMIACT# VCC2 AG
AE VCC3 D/P# A23 PCHK# VSS AF
AD VSS INTR APCHK# PBREQ# VCC2 AE
AC VCC3 R/S# NMI PBGNT# VSS AD
AB VSS SMI# PRDY PHITM# VCC2 AC
AA VCC3 IGNNE# INIT HOLD VSS AB
Z VSS PEN# WB/WT# PHIT# VCC2 AA
Y VCC3 FRCMC# BF0 BOFF# VSS Z
X VSS BF1 NA# BRDYC# VCC2 Y
W VCC3 BF2 NC BRDY# VSS X
V VSS STPCLK# KEN# EWBE# VCC2 W
U VCC3 VSS VCC3 AHOLD VSS V
T VSS VCC3 INV CACHE# VCC2 U
S VCC3 NC NC M/IO# VSS T
R VSS NC BP3 BP2 VCC2 S
Q VCC3 CPUTYP TEST# PM1/BP1 VSS R
P VSS TMS FERR# PM0/BP0 VCC2 Q
N VCC3 TDI TDO IERR# VSS P
M VSS TCK DP7 D63 VCC2 N
L VCC3 PICD1 VCC3 D62 VSS M
K VSS D0 D60 D61 VCC2 L
J VCC3 D2 PICD0 D59 VSS K
H VSS PICCLK D58 D57 VCC2 J
G VCC3 D1 D3 D56 VSS H
F D4 D5 D53 D55 VCC2 G
E VSS A25 A11 DP5 D51 DP6 F
D DP0 D8 D12 DP1 D19 D23 D26 D28 D30 DP3 D33 D35 D37 D39 D40 D44 D48 D50 E
C D9 D10 D14 D17 D21 D24 DP2 D25 D27 D29 D31 D32 D34 D36 D38 DP4 D45 D47 INC D
B D11 D13 D16 D20 VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS VSS D43 INC C
A NC D15 D18 D22 VCC3 VCC3 VCC3 VCC3 VCC3 VCC3 VCC2 VCC2 VCC2 VCC2 VCC2 VCC2 D41 INC B
A
37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Pin summary

A20M#	I	**Address bit 20 Mask** is an emulator signal.
A31..A3	IO	**Address**, in combination with **byte enable**, indicate the physical addresses of memory or device that is the target of a bus transaction. This signal is an output.

		when the processor is initiating the bus transaction, and an input when the processor is receiving an inquire transaction or snooping another processor's bus transaction.
ADS#	IO	**ADdress Strobe**, when asserted, indicates new bus transaction by the processor, with valid **address** and **byte enable** simultaneously driven.
ADSC#	O	**Address Strobe Copy** is driven identically to **address strobe**
AHOLD	I	**Address HOLD**, when asserted, causes the processor to cease driving **address** and **address parity** in the next bus clock cycle.
AP	IO	**Address Parity** contains even parity on the same cycle as **address**. **Address parity** is generated by the processor when **address** is an output, and is checked when **address** is an input. A parity error causes a bus error machine check.
APCHK#	O	**Address Parity CHecK** is asserted two bus clocks after EADS# if **address parity** is not even parity of **address**.
APICEN	I	**Advanced Programmable Interrupt Controller ENable** is not implemented.
BE7#..BE0#	IO	**Byte Enable** indicates which bytes are the subject of a read or write transaction and are driven on the same cycle as **address**.
BF1..BF0	I	**Bus Frequency** is sampled to permit software to select the ratio of the processor clock to the bus clock.
BOFF#	I	**BackOFF** is sampled on the rising edge of each bus clock, and when asserted, the processor floats bus signals on the next bus clock and aborts the current bus cycle, until the backoff signal is sampled negated.
BP3..BP0	O	**BreakPoint** is an emulator signal.
BRDY#	I	**Bus ReaDY** indicates that valid data is present on **data** on a read transaction, or that **data** has been accepted on a write transaction.
BRDYC#	I	**Bus ReaDY Copy** is identical to BRDY#; asserting either signal has the same effect.
BREQ	O	**Bus REQuest** indicates a processor initiated bus request.
BUSCHK#	I	**BUS CHecK** is sampled on the rising edge of the bus clock, and when asserted, causes a bus error machine check.
CACHE#	O	**CACHE**, when asserted, indicates a cacheable read transaction or a burst write transaction.
CLK	I	bus **CLocK** provides the bus clock timing edge and the frequency reference for the processor clock.
CPUTYP	I	**CPU TYPe**, if low indicates the primary processor, if high, the dual processor.

183

D/C#	I	**Data/Code** is driven with the address signal to indicate data, code, or special cycles.
D63..D0	IO	**Data** communicates 64 bits of data per **bus clock**.
D/P#	O	**Dual/Primary** is driven (asserted, low) with **address** on the primary processor
DP7..DP0	IO	**Data Parity** contains even parity on the same cycle as **data**. A parity error causes a bus error machine check.
DPEN#	IO	**Dual Processing Enable** is asserted (driven low) by a Dual processor at reset and sampled by a Primary processor at the falling edge of reset.
EADS#	I	**External Address Strobe** indicates that an external device has driven **address** for an inquire cycle.
EWBE#	I	**External Write Buffer Empty** indicates that the external system has no pending write.
FERR#	O	**Floating point ERRor** is an emulator signal.
FLUSH#	I	**cache FLUSH** is an emulator signal.
FRCMC#	I	**Functional Redundancy Checking Master/Checker** is not implemented.
HIT#	IO	**HIT** indicates that an inquire cycle or cache snoop hits a valid line.
HITM#	IO	**HIT to a Modified line** indicates that an inquire cycle or cache snoop hits a sub-block in the M cache state.
HLDA	O	**bus HoLD Acknowlege** is asserted (driven high) to acknowlege a **bus hold request**
HOLD	I	**bus HOLD request** causes the processor to float most of its pins and assert **bus hold acknowlege** after completing all outstanding bus transactions, or during reset.
IERR#	O	**Internal ERRor** is an emulator signal.
IGNNE#	I	**IGNore Numeric Error** is an emulator signal.
INIT	I	**INITialization** is an emulator signal.
INTR	I	**maskable INTeRrupt** is an emulator signal.
INV	I	**INValidation** controls whether to invalidate the addressed cache sub-block on an inqure transaction.
KEN#	I	**Cache ENable** is driven with **address** to indicate that the read or write transaction is cacheable.
LINT1..LINT0	I	**Local INTerrupt** is not implemented.
LOCK#	O	**bus LOCK** is driven starting with **address** and ending after **bus ready** to indicate a locked series of bus transactions.
M/IO#	O	**Memory/Input Output** is driven with **address** to indicate a memory or I/O transaction.
NA#	I	**Next Address** indicates that the external system will accept an **address** for a new bus cycle in two bus clocks.
NMI	I	**Non Maskable Interrupt** is an emulator signal.